

The joy of Smalltalk

An introduction to Smalltalk

© Ivan Tomek, 2000

Table of contents

	Preface
Part 1	Introduction to object-oriented problem solving and the Smalltalk language
	Chapter 1 - Object-oriented problem solving - essential concepts
	Chapter 2 - Finding objects
	Chapter 3 - Principles of Smalltalk
Part 3	Essential classes, user interface components, and application development
	Chapter 4 - True and False objects, blocks, selection, iteration
	Chapter 5 - Numbers
	Chapter 6 - Design of applications with user interfaces, Action Buttons
	Chapter 7 - Introduction to collections, Table widgets
	Chapter 8 - More sequenceable collections, List and menu widgets
	Chapter 9 - Unordered collections - sets, bags, and dictionaries
	Chapter 10 - Streams, files, BOSS
Part 3	Advanced topics
	Chapter 11 - Stacks, queues, linked lists, and trees
	Chapter 12 - More about user interfaces
	Chapter 13 - Processes and their coordination
	Appendices
	Appendix 1 - Check Boxes, Radio Buttons, Input Fields, and their applications
	Appendix 2 - Dataset, Notebook, Subcanvas, Dialog Window, Menus
	Appendix 3 - Chess board – a view holder application
	Appendix 4 - Classes and Metaclasses
	Appendix 5 - Style recommendations
	Appendix 6 - Projects
	Appendix 7 - Smalltalk syntax
	Appendix 8 - Smalltalk tidbits
	Appendix 9 - Selected Smalltalk products
	Glossary
	References
	Index

Detailed Table of Contents

Preface

Part 1 Object-oriented problem solving essential concepts

Chapter 1 - Object-oriented problem solving - essential concepts

- 1.1 Introduction
- 1.2 What is object-oriented problem solving?
- 1.3 Examples of objects in computer applications
- 1.4 How does an object-oriented computer application work?
- 1.5 Classes and their instances
- 1.6 A first look at Smalltalk classes
- 1.7 Object properties
- 1.8 Using System Browser to find out about objects
- 1.9 Class, subclass, superclass, abstract class, inheritance, class hierarchy
- 1.10 Smalltalk's class hierarchy
- 1.11 Polymorphism
- Conclusion

Chapter 2 - Finding objects

- 2.1 Examples of object-based solutions
- 2.2 Finding objects
- 2.3 Example 1 –A Rental Property Management Program
- 2.4 Example 2 – The Farm Program
- Conclusion

Chapter 3 - Principles of Smalltalk

- 3.1 Basic rules
- 3.2 Maintaining access to objects - variables
- 3.3 Writing and executing programs
- 3.4 More about variables
- 3.5 Smalltalk messages
- 3.6 Nesting of expressions
- 3.7 Order of evaluation of messages
- 3.8 Tracing message evaluation with the Debugger
- 3.9 Cascading
- 3.10 Global variables, class instance variables, and pool dictionaries
- Conclusion

Part 2 Essential classes, user interface components, and application development

Chapter 4 - True and False objects, blocks, selection, and iteration

- 4.1 Why we need true and false objects
- 4.2 Boolean messages for deciding whether to take an action or not
- 4.3 The definition of ifTrue:
- 4.4 Selecting one of two alternative actions
- 4.5 Use ifTrue: and ifFalse: only when necessary
- 4.6 Creating a new class and a method
- 4.7 Logic operations
- 4.8 Exclusive or, equality, equivalence
- 4.9 Use of Booleans to repeat a block of statements
- 4.10 Other forms of iteration
- Conclusion

Chapter 5 - Numbers

- 5.1 Numbers
- 5.2 Operations on numbers
- 5.3 Implementation of binary arithmetic messages - double dispatching and primitives

- 5.4 Using numbers for iteration - 'repeat n times'
- 5.5 Repeating a block for all numbers between a start and a stop value
- 5.5 Repeating a block with a specified step
- 5.7 Measuring the speed of arithmetic and other operations
- 5.8 Declaring a new class: Currency
- 5.9 Another implementation of Currency
- 5.10 Generalized rectangles
- Conclusion

Chapter 6 - Design of Applications with graphical user interfaces

- 6.1 Example of application development: An application selector
- 6.2 Implementing the user interface - the window
- 6.3 Painting widgets and defining their properties
- 6.4 Defining *Action* and *Aspect* properties
- 6.5 The remaining *Action* methods
- 6.6 Text Editor widget
- 6.7 Value holders, models, and dependents
- 6.8 Opening an application - hook methods
- 6.9 MVC – the Model – View – Controller triad
- 6.10 IDs make widgets accessible at run time - a Tic-Tac-Toe game
- Conclusion

Chapter 7 - Introduction to Collections

- 7.1 Introduction
- 7.2 Essential collections
- 7.3 Properties shared by all collections
- 7.4 Arrays
- 7.5 Examples of uses of arrays
- 7.6 Two-dimensional arrays - tables and matrices
- 7.7 Implementing an n-dimensional array
- 7.8 Use of TwoDList in the Table widget
- Conclusion

Chapter 8 - More sequenceable collections, List widgets

- 8.1 Class OrderedCollection
- 8.2 Several examples with ordered collections
- 8.3 Class SortedCollection
- 8.4 Ordered collections as the basis of dependence
- 8.5 Tennis – another example of dependency
- 8.6 The List collection
- 8.7 String, Text, and Symbol - an introduction
- 8.8 Text - its nature and use
- 8.9 List widgets
- Conclusion

Chapter 9 - Sets, bags, and dictionaries

- 9.1 Sets
- 9.2 Copying objects
- 9.3 Bags
- 9.4 Associations and dictionaries
- 9.5 Dictionary with multiple values
- 9.6 Example - a two-way dictionary
- 9.7 A Finite State Automaton
- Conclusion

Chapter 10 - Streams, files, and BOSS

- 10.1 Introduction to streams
- 10.2 Internal streams
- 10.3 Examples of operations on internal streams
- 10.4 Example: A text filter

- 10.5 Example: Circular Buffer
- 10.6 Introduction to files and external streams
- 10.7 Class Filename
- 10.8 Examples of file operations that don't require external streams
- 10.9 External streams
- 10.10 Storing objects with BOSS
- 10.11 Other ways of storing objects
- Conclusion

Part 3 Advanced topics

Chapter 11 - Stacks, queues, linked lists, trees, and graphs

- 11.1 Stack - an access-at-top-only collection
- 11.2 Context Stack and Exceptions
- 11.3 More about exceptions
- 11.4 Queues
- 11.5 Text Filter – a new implementation
- 11.6 Linked Lists
- 11.7 Trees
- 11.8 Use of trees in compilation
- 11.9 Graphs
- Conclusion

Chapter 12 - Developing user interfaces

- 12.1 Principles of user interfaces – display surfaces, graphics contexts, and visual parts
- 12.2 An example of the use of windows – a virtual desktop
- 12.3 Principles of displaying – graphics contexts, geometric objects, and other concepts
- 12.4 Images, pixmaps, masks, and paint
- 12.5 Models, views, and controllers revisited
- 12.6 Creating UI components with the view holder widget
- 12.7 Controllers

Chapter 13 - Processes and their coordination, additional UI topics

- 13.1 A stopwatch and the concept of a Process
- 13.2 Alarm tool
- 13.3 Coordinating mutually dependent processes – train simulation
- 13.4 Making train simulation layout customizable
- Conclusion

Appendices

Appendix 1 - Check Boxes, Radio Buttons, Input Fields, and their applications 32 pages

- A.1.1 Check Boxes and Radio Buttons - an introduction
- A.1.2 Check Boxes
- A.1.3 Radio Buttons
- A.1.4 Input Fields
- A.1.5 A computerized restaurant menu
- A.1.6 Other implementations of restaurant menu
- A.1.7 Validation of user input
- A.1.8 A course evaluation program
- A.1.9 A (very) simple computerized Tax Form
- Conclusion

Appendix 2 - Dataset, Subcanvas, Notebook, Dialog Window, Menus

- A.2.1 Dataset widgets
- A.2.2 Subcanvas
- A.2.3 Diary - Using a subcanvas to reuse a complete application
- A.2.4 The Notebook widget
- A.2.5 Dialog windows

- A.2.6 Menus in general and Pop up Menus in particular
- A.2.7 Menu Bars
- Conclusion

Appendix 3 - Chess board – a custom user interface

- A.3.1 Chess - specification
- A.3.2 Preliminary design
- A.3.3 Design refinement
- A.3.4 Implementation
- Conclusion

Appendix 4 - Classes, Metaclasses, and Metaprogramming

- A.4.1 Classes and Metaclasses
- A.4.2 What is the complete class hierarchy?
- A.4.3 What are the main properties of metaclasses?
- A.4.4 Class Behavior
- A.4.5 Class ClassDescription
- A.4.6 Class Class
- A.4.7 Is this magic useful?
- A.4.8 Another example of metaprogramming: Enhanced Workspace
- A.4.9 Another example: Wrapping objects to intercept messages
- Conclusion

Appendix 5 - Style recommendations

- A.5.1 Introduction to Smalltalk style guidelines
- A.5.2 Naming
- A.5.3 Comments
- A.5.4 Names of common protocols
- A.5.5 Introduction to idioms and patterns
- A.5.6 General patterns
- A.5.7 Methods
- A.5.8 Behaviors
- A.5.9 Variables

Glossary
References
Index

Preface

The aim of this textbook

The goal of this textbook is to introduce concepts of object-oriented software development and the programming language Smalltalk as a vehicle for their implementation.

Why object-oriented software development

After the introduction and experience with a series of programming paradigms, object-oriented (OO) software development has become the undisputed mainstay approach to the development of modern software. The reason for this is that the OO paradigm best satisfies demands for fast development of reliable and maintainable software. Its principle of viewing software as modules modeling real world objects is a natural reflection of both clients' and developers' view of the world, and provides a basis for building extendible and reusable libraries of individual objects and whole frameworks that can be relatively efficiently extended or customized to build new applications.

Why Smalltalk

Besides being the first full-fledged commercial object-oriented programming language, Smalltalk is one of the most exciting and powerful programming languages. The fact that most programmers who learn it prefer never to use any other language is a testimony to this. Its uncompromising reliance on OO principles and its basic simplicity also mean that if you want to learn object-oriented programming, there is nothing in Smalltalk to distract you. In Smalltalk you cannot do anything without objects and the current shift towards object-oriented programming thus makes Smalltalk the perfect vehicle for the development of a sound OO perspective. Very few languages offer such an undiluted environment. Smalltalk implementations are also very interactive and this makes Smalltalk an ideal tool for experimentation and hands-on classroom presentations.

Another interesting feature of Smalltalk is that it is not only a language but also a complete programming environment with tools such as browsers, object inspectors, debuggers, and file editors. While this is not a rarity any more, Smalltalk still distinguishes itself by the fact that it contains all the source code of its environment and its own implementation, allowing the user to study the principles of sophisticated OO design, and to make personal modifications to the environment if desired.

Since the implementation of a program development environment requires most of the building modules that an application requires, all the essential building blocks already in the environment's library. Since libraries of other OO language have been largely derived from Smalltalk, understanding the Smalltalk library is also an excellent introduction to libraries of other OO languages as well. The simplicity of Smalltalk makes the study of its architecture much easier than it ever could be in most other languages.

Because the environment contains its source code, you can learn much about Smalltalk directly from its implementation. Since the source code includes the compiler, the programming tools, user interfaces, and operating system components, Smalltalk is also excellent for learning about most software-related issues of Computer Science. This includes algorithms and data structures, data base management systems (both relational and object-oriented), operating systems, compilers, networks, graphics, simulation, and others. Whole Computer Science curricula can be based on Smalltalk.

Those with a pragmatic outlook will appreciate that Smalltalk is currently, after C++, the second most popular object-oriented language for large software projects, both stand-alone and network-based. It is also the second fastest growing object-oriented programming language after Java. As a consequence, there is a considerable shortage of qualified Smalltalk programmers and mastering the language is a very good investment.

Why has Smalltalk been so popular in commercial applications? The main advantages of Smalltalk over other languages are its conceptual simplicity and its undisputed superiority for prototyping and rapid application development. The high productivity of Smalltalk development is due to the ease with which new

Smalltalk code be implemented and tested which is directly related to its design principles, and to the very large library of reusable and easily extendible built-in parts. Smalltalk's excellent support for user interface development is another major reason for its popularity.

The most popular Smalltalk implementations are also very portable among many hardware and operating system platforms. As an example, the VisualWorks Smalltalk applications developed in this book will run without any changes on 13 different platforms including Microsoft Windows, OS/2, the Macintosh and the SUN. It is even possible to develop different parts of a non-trivial Smalltalk application on different platforms and merge them together. This high degree of portability is a great economical advantage not available in most other languages.

An important consequence of the fact that Smalltalk is so popular is that its developers must keep up to date with the latest technological advances. As a result, Smalltalk was, for example, one of the first languages to provide advanced extensions to distributed objects. It is also the language in which some of the most powerful and popular development environments for other languages such as VisualAge Java and C++ have been developed.

Finally, developers appreciate Smalltalk because it is very mature - it has been in existence for almost 30, and in productive use for almost 20 years. This is almost unbelievable when you consider that even after all these years it remains one of the most progressive programming environments available. The OO principles of all major OO languages in use today have been built on the basis of the Smalltalk experience and are still borrowing principles from it, without ever quite achieving its power and simple elegance, in our opinion.

How this book differs from other Smalltalk books

Although there are many excellent texts about Smalltalk, this is the only one that covers all the following topics

- OO concepts
- OO application development
- Smalltalk programming environment including its user interface painting tools
- Smalltalk base classes
- elements of Smalltalk style

illustrates them on many examples, both small and large, and provides numerous exercises and projects that challenge the reader to master the material by personal experimentation. You can download the software presented in the text from the author's home page at <http://www.acadiau.ca/ivan>.

The book does not assume any previous programming background but the subject matter is not limited to essentials and provides enough depth to allow the reader to write substantial applications and to understand the deeper principles of Smalltalk. There is, however, no way that one could write a book of this size and introduce cover all functionality of Smalltalk. We thus acknowledge that in spite of the breadth of our coverage, many essential topics that are not covered in great depth (such as exceptions or more complicated application architectures) or that remain uncovered. As an example, we have not touched the interface between Smalltalk and databases, the use of Smalltalk for network programming, interfacing Smalltalk to external applications, and other subjects. We also have not covered all graphical user interface widgets, code stripping, team development tools, and other important but more advanced subjects.

Besides the unparalleled completeness of coverage, the structure of this book is also unusual. We begin with a presentation of the principles of the object-oriented paradigm, continue with a rather detailed introduction to object-oriented analysis and design, and then present both the essential components of the Smalltalk library and its user interface building tools and components. User interfaces are introduced rather early and in relative detail because we think that in this age of sophisticated computer applications one should build programs that not only do interesting things but also provide a pleasing user interface. Besides, creating programs with modern user interfaces increases the enjoyment of programming and makes its study more stimulating. And Smalltalk makes this rather easy.

The scope of this book and its intended audience

Since the book does not make any assumptions about the reader's background, it is suitable for newcomers to Smalltalk and even those who never studied programming. In fact, we use the text in a first year programming course for Computer Science students. However, since the scope goes well beyond essentials the book is also of interest to those who already know another programming language and want to learn object-oriented programming or Smalltalk.

Readers who already know Smalltalk will also find the book useful because the presentation is quite different from that of other books and includes material not found elsewhere. This includes examples of implementation of conventional abstract data types, principles of implementation of customized user interfaces, processes, and a discussion of metaclasses with examples of metaprogramming. Besides, Smalltalk is always worth looking at from a different perspective because there is never an end to learning something new about this rich and flexible programming environment.

How to use this book

Whether you are using this book in a course or for self-study, we recommend the following approach: Study the material, try the examples in the text, and do as many exercises as possible to become comfortable with Smalltalk and its environment and program development skills. As you progress, implement some of the projects listed in the Appendix 6 using the approach presented in the main text.

Which Smalltalk

Smalltalk is currently undergoing standardization and many of the classes in its library are now required in all compliant products. However, there is no way to prescribe all details of an inherently extensible environment such as Smalltalk. As a consequence, individual Smalltalk dialects will always differ from one another in implementation, design, user interfaces, and even conceptual models of its building blocks. From the point of view of a Smalltalk programmer, especially a novice, the main differences between the dialects are in the user interface of the programming environment and the number and implementation of the built-in user interface components. Support for graphics, the structure of windows and their components, and support for file operations also differ from one dialect to another.

Because of the differences between different implementations, every Smalltalk book that goes beyond basics must select one particular dialect and cover language-dependent features as implemented in this dialect. We chose VisualWorks because this happens to be the dialect that we have used for the longest time and because it is the dialect most directly related to original Smalltalk. If you don't use VisualWorks Smalltalk, your user interface will be somewhat different from the illustrations in this book, some tools such as user interface painters will be different, and the corresponding classes will be implemented or organized differently. However, most of the material presented in this book applies to any Smalltalk dialect without any change. See Appendix 8 for a listing of commercial and free Smalltalk products available at the time of this writing.

Acknowledgments

Like many authors who wrote about Smalltalk before me, I want to express my thanks and admiration to the group of visionaries who invented Smalltalk and to those who keep extending it creatively and applying it to new domains.

I am very grateful to the many readers of the preliminary forms of the manuscript who taught me a lot by their criticism and suggested ideas that are now implemented in the text. I am also obliged to my students whose blank looks alerted me to inappropriate presentation strategies and whose improvements of classroom examples served to improve the text. I hope that I will get similar constructive suggestions for improvements from the readers of this book as well. Finally, I must acknowledge the influence of the Smalltalk newsgroup on Internet at comp.lang.smalltalk. I learned a lot from it and borrowed some of the ideas expressed at this forum for examples and exercises.

I have been teaching Smalltalk for the last five years and forgotten many of the students who actively shaped my understanding of Smalltalk and influenced my teaching, but I wish to thank the following students and friends whose names stand out in my memory. They include (alphabetically) Fauzi

Ali, James Benedict, Peter Burka, Randy Giffen, Kenn Hussey, James Moody, Oliver Oey, Mark Rhodenizer, Ravi Palepu, and Donald Smith. Most of them are now employed as professional Smalltalk programmers.

Finally, but most importantly, I wish to thank my family. During the long time that it took to write this book, my wife Jana managed to pretend that spending the time that other people devote to holidays on my computer was a part of normal life, smoothed over my continuous swings between fascination and frustration with this book, and added a human dimension to my life. Our three children - Ivan, Ondrej, and Dominika – helped greatly to maintain this illusion. I have been very lucky to have a family that provided such a warm, supportive, and stimulating environment.

Chapter 1 - Object-oriented programming - essential concepts

Overview

In this chapter, we will first show what Smalltalk looks like and then introduce the essential concepts of object-oriented problem solving and illustrate them on several examples. Some of these examples are real-world situations, others are taken from Smalltalk itself.

The principle of object-oriented problem solving is the insight that many problems are best approached by constructing models of real-world situations. The basis of these models are interacting objects with well-defined properties and behaviors. Solving a problem using the object-oriented approach thus consists of identifying appropriate objects and describing the functions that they must be able to perform and the information that they must hold. A computer application can then be constructed by converting such a description into a programming language. Programming languages that provide facilities for constructing such descriptions are called object-oriented and Smalltalk is one their prime examples.

1.1 Introduction

Since you are probably eager to start writing and executing programs, we will begin with a few examples of Smalltalk code.

Example 1: Arithmetic operations

The line

$(15 * 19) + (37 \text{ squared})$

is a typical Smalltalk arithmetic expression and you can see that it multiplies two numbers and adds the result to the square of a third number. To test this code, start VisualWorks Smalltalk using the instructions given in your User's Guide¹. You will get a screen containing the VisualWorks launcher window shown in Figure 1.1. Click the Workspace button and VisualWorks will open the window shown on the left in Figure 1.2.

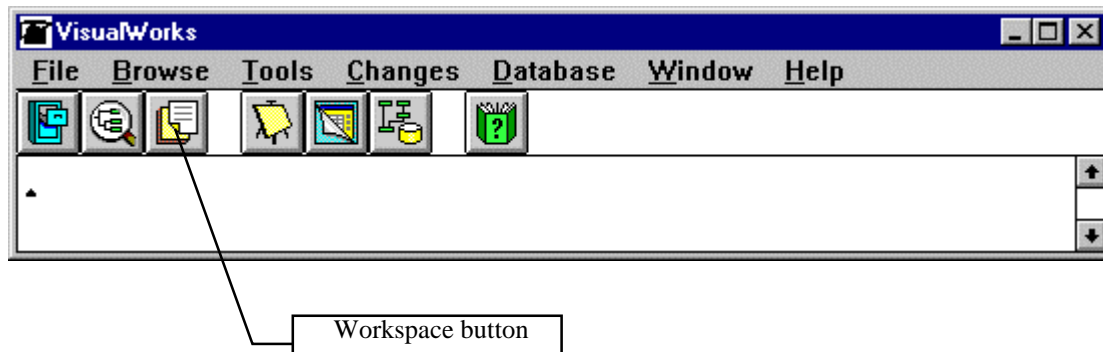


Figure 1.1. VisualWorks launcher window.

Enter the example code into the Workspace as if you were using a word processor as follows: Click the left mouse button inside the Workspace and enter the text making sure that it looks exactly like our example. Then 'select' the text as you would in a word processor. Smalltalk provides several selection

¹ All our illustrations use VisualWorks Smalltalk. Other Smalltalk dialects have different user interfaces and their extended libraries are different. Most of the features covered in this book however apply.

shortcuts. For example, clicking twice at the beginning or at the end of the text 'view' of a Smalltalk selects all text in the view, and clicking twice at the beginning or at the end of a line selects the whole line.² Press the left button just before the start of the text and drag the cursor to the right across the text, releasing the button at the end. The text is now highlighted as in the center of Figure 1.2. Finally, press and hold down the second mouse button from the left and click the *print it* command in the displayed pop up menu². Smalltalk will execute the code and print the result in the Workspace as in the right window in Figure 1.2.

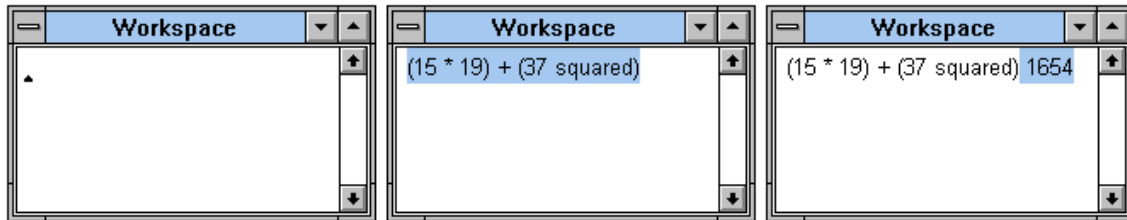


Figure 1.2. Workspace window: initial state (left), with selected text of Example 1 (middle), and displaying the result (right).

Example 2: Comparison of results of numeric expressions

The line

```
(1327 squared) < (153 * 20000)
```

is a typical test to determine whether a comparison of two expressions gives a yes or a no answer. Type it into the Workspace, select it³, and execute it with *print it*. The answer displayed in the Workspace will be either true or false because Smalltalk treats a comparison as a question: 'Is it true that ...?'

Example 3: String comparison

The line

```
'abc' < 'xyz'
```

is a typical string comparison that determines whether the string on the left precedes the string on the right in alphabetical ordering. Guess what the result should be and test whether you guessed right.

Example 4. A string operation

As you are beginning to see, Smalltalk is designed to be easy to read. In fact, one of the original goals of Smalltalk designers was to create a programming language that even children could use. What do you think is the result of the following expression?

```
'Smalltalk' asUppercase
```

Test whether you guessed right.

Example 5. Simple output

² Since the leftmost button of the mouse is used to make selections, Smalltalk programmers call it the <select> button. The next button displays a popup menu with operation commands such as print it and it is thus called the <operate> button.

³ Smalltalk provides several selection shortcuts. For example, clicking twice at the beginning or at the end of the text view selects all text in the view, and clicking twice at the beginning or at the end of a line selects the whole line.

All Smalltalk code consists of 'messages' to 'objects' and this is why it is called object-oriented. Some of the messages used above include `squared`, `factorial`, and `<` and the objects include 27, 13, and 'abc'. The Smalltalk library includes thousands of messages and you can easily create any number of your own, but all of them have one of three possible structures. You have seen two types of messages above (messages consisting of a single word such as `squared`, and messages consisting of a special symbol such as `<`), and this example introduces the only kind of message available in Smalltalk that you have not yet encountered. This kind of message is distinguished by the fact that its name is followed by a colon and an 'argument':

Transcript show: 'Hi there!'

prints the text Hi there in the Transcript, the text area at the bottom of the launcher window (Figure 1.3).

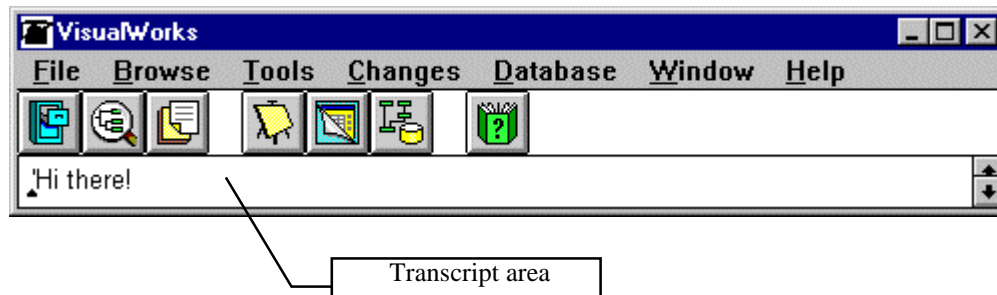


Figure 1.3. Result of evaluating Transcript show: 'Hi there!' with *print it*.

Example 6. A more complicated comparison

Expression

567835 between: (27 squared) and: (13 factorial)

tests whether 567835 is between the values of 27 squared and 13 factorial. Guess the result and test whether the expression 'returns' true or false. It shows a messages of the third kind but this one has two word-colon pairs (called 'keywords') instead of one. There is no practical limit on the number of keywords that a message may have.

To summarize our brief experience, *Smalltalk treats everything as messages sent to objects* and its interpretation of expressions such as

15 squared

is as follows: *Take the Number object 15 and send it the message squared*. All number objects understand the message `squared` and when object 15 gets the message, it calculates its square and returns this Number object as the answer. Similarly, when Smalltalk executes

'Smalltalk' asUppercase

it treats the text string 'Smalltalk' as a string object, and `asUppercase` as a message to it, and returns the string object 'SMALLTALK'.

The more complicated expression

(15 * 19) + (37 squared)

is executed in several steps. First, Smalltalk asks object 15 to execute message `*` with argument 19. This returns object 285, the product of 15 and 19. The original code has now, in effect, been reduced to

285 + (37 squared)

Smalltalk now asks 37 to execute message squared. This returns object 1369 and the code now effectively becomes

285 + 1369

Smalltalk now asks object 285 to execute message + with argument 1369. This returns the final result 1654. Note that this works because *all Smalltalk messages return objects*, and messages can thus be combined.

The examples that you have just seen cover all possible forms of Smalltalk messages and if they give you the impression that Smalltalk is essentially simple, you are quite right - Smalltalk is based on very few very powerful ideas. Unfortunately, the ease of reading and writing pieces of Smalltalk programs does not mean that writing significant Smalltalk programs such as a word processors or spreadsheets is very easy. Difficult task are difficult even in Smalltalk - but not as difficult as in most other languages as you will see later.

At this point, you would probably like to proceed to other Smalltalk programs, and if you really cannot resist it, you can skip to Chapter 3 which starts our discussion of Smalltalk. However, the ideas of an object and a message have some non-obvious implications and properties and we strongly suggest that you now complete this chapter to find out about some very important general concepts of object-oriented programming.

It will probably not surprise you if we also suggest that you then continue with Chapter 2 which is dedicated to the principles of finding the right kinds of objects for your application. The reason why this is important is that developing object-oriented applications requires that you first identify the right objects for your task and *then* write the code describing them in your programming language. Programming in Smalltalk thus requires two skills - familiarity with Smalltalk, and ability to find the objects that can be combined to solve your problem. Both of these skills are essential: If you know how to use Smalltalk but cannot design the proper objects for your application, you cannot even start writing programs. And if you can design but don't know Smalltalk, you obviously cannot convert your paper design into working Smalltalk code. This means that we now have a choice: Explain the Smalltalk language first and object-oriented design next, or vice versa. We decided to start with principles and design because learning about design greatly improves understanding of Smalltalk and contributes to Smalltalk skills.

Main lessons learned:

- Smalltalk is a very readable language in which all code is interpreted as messages to objects. Languages based on this principle are called object-oriented.
- Although the object-oriented principle is very simple, it has some subtle implications and consequences that must be understood before one can start writing significant programs.
- Program development requires two skills: the knowledge of a programming language and the ability to find suitable objects that can be used to solve the problem.
- The Visual Launcher is the opening window in VisualWorks Smalltalk. All other tools can be accessed from it.
- The left mouse button is called the <select> button. Use it to select text and other items. The button next to it is called the <operate> button and pressing it opens a pop up menu with commands.

Exercises

1. Execute the examples from this section in Smalltalk and explore the environment on your own.

1.2 What is object-oriented problem solving?

When you think about a problem such as planting a garden, you think of objects (a spade, a wheelbarrow, a watering can, and various kinds of plants) and the things that they can do (a spade can dig a hole, a watering can can be filled and emptied, and plants can be planted). The principle of object-oriented problem solving is the same: Think of solutions in terms of the required objects and their functionality, and the relation between all objects that participate in the solution. When put together, the program then works as follows: To perform a task, send a message to an appropriate object which directly executes those operations that it can handle, and sends messages requesting help to other objects for operations that are beyond its area of expertise. In this and the following sections, we will now illustrate this concept on several simple examples taken from programming and non-programming situations.

World 1: Airport

Consider an airport with travelers, arriving and departing planes, and the necessary ground support. One of the possible situations that we might wish to handle can be described by the following scenario:

Scenario: Getting a taxi

Mr. Brittle's airplane is landing at Terminal 3B of the International Airport in Halifax and he wants to book a taxi to take him to the downtown. He turns on his laptop computer and negotiates the taxi with a taxi dispatcher via an e-mail conversation along the following lines:

Conversation:

1. Mr. Brittle - message to dispatcher: 'My plane is landing and I need a taxi at Terminal 3B. My name is Brittle'.
2. Dispatcher - message to a free taxi: 'Please go to Terminal 3B and wait for Mr. Brittle'.
3. Taxi driver - replies to dispatcher: 'I am on my way'.
4. Dispatcher - replies to Mr. Brittle: 'The taxi is on its way'.

The flow of the conversation is illustrated graphically in Figure 1.3. In this diagram, time is running from the top down, columns are labeled with names of participants, full lines with arrows indicate who is communicating with whom and what the communication is about, dotted lines represent confirmations of message completion.

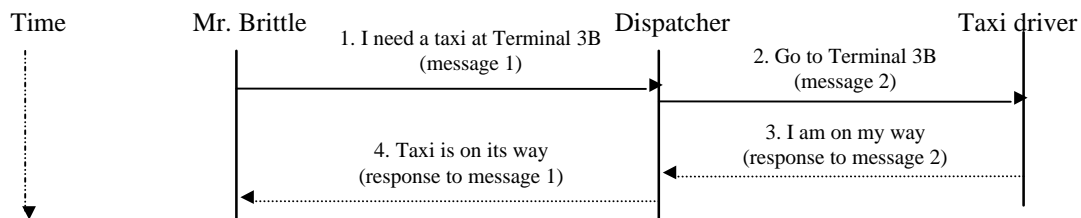


Figure 1.3. Graphical representation of Scenario 1.

The 'objects' participating in this scenario and their communication are shown in Figure 1.4. They include Mr. Brittle, the dispatcher, and the taxi driver, and the conversation involves two messages (one

from Mr. Brittle to the dispatcher and one from the dispatcher to the driver) and confirmation of their successful execution.

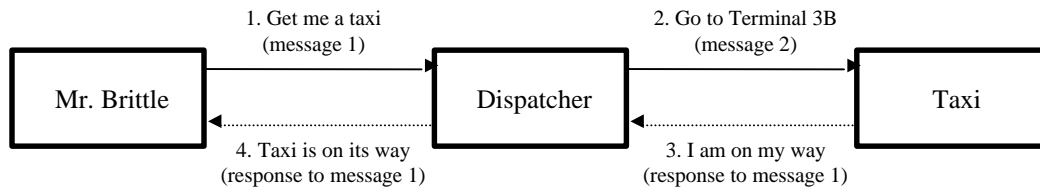


Figure 1.4. Another representation of Conversation 1.

The real airport is, of course, more complex and many other scenarios and conversations are possible. As an example, the dispatcher might inform Mr. Brittle that there is no taxi, the taxi driver might tell the dispatcher that the car is broken, access ramp of Terminal 3B might be closed, and so on. Our script is only one example presented to illustrate the nature of object-oriented thinking and the remaining examples are presented in the same spirit.

World 2: Household management

This example shows how I could deal with a little emergency in my household.

Scenario: Taking care of unexpected visitor

My sister Monica has unexpectedly arrived with her two children and I don't have anything to offer them. Monica likes roses and chocolate cake, her children like hamburgers. Having no time to go to a store, I call a catering company. The conversation might go as follows.

Conversation:

1. I - message to caterer: 'Please deliver a bunch of roses, a chocolate cake, and two hamburgers'.
2. Caterer to florist: 'Please deliver a bunch of roses'.
3. Florist to caterer: Delivers roses to caterer.
4. Caterer to MacDonald's: 'Please deliver two hamburgers'.
5. MacDonald's to caterer: Delivers two hamburgers to caterer.
6. Caterer to Tim Horton's: 'Please deliver a chocolate cake'.
7. Tim Horton's to caterer: Delivers chocolate cake to caterer.
8. Caterer to me: Delivers rose, cake, and hamburgers to me.

The object-oriented view of this situation (Figure 1.5) has five participants, or actors: me, the caterer, the florist, MacDonald's, and Tim Horton's. Communications 1, 2, 4, and 6 are messages, communications 3, 5, 7, and 8 confirm their execution and deliver the requested objects. The responsibilities of participating actors - the services that they are expected to be able to perform - include the ability to satisfy requests for party items (caterer), flowers (florist), and so on.

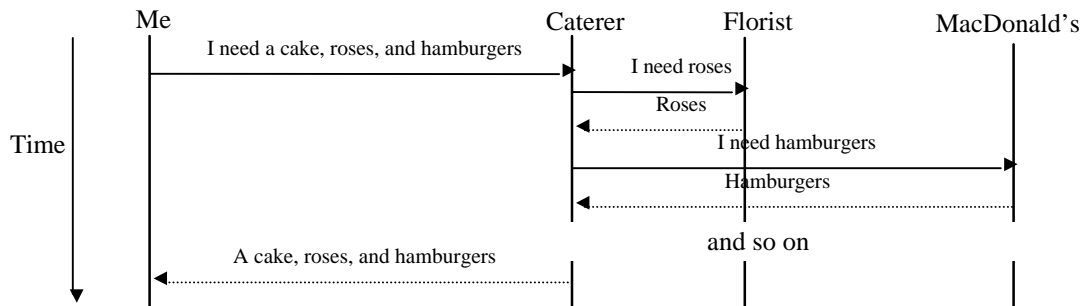


Figure 1.5. Partial script of Scenario 2.

We can now make several conclusions concerning the object-based view of problems:

- Problems can be solved by asking specialized objects to perform tasks within the scope of their expertise.
- To ask an object to perform an operation, send it a message.
- Even complex tasks can be executed by collaboration among specialized objects.
- When an object receives a message, it executes those parts of the task that it can handle and sends messages to other objects to accomplish those tasks that are outside its area of expertise.
- An object's response to a message can be formulated as follows: Every message returns an object. The returned object may be a physical object such as a hamburger, or just a piece of information such as computer data or a confirmation that a message has been completed.

Executing a task by asking another object to execute it is called *delegation*. In our example, the caterer delegates the task of procuring roses to the florist - the florist is the caterer's *collaborator*. The chain of messages delegating sub-tasks to collaborators may be several levels deep.

World 3: Microwave oven control

My microwave oven provides many functions performed by using the panel in Figure 1.6. I will now describe one of its uses.

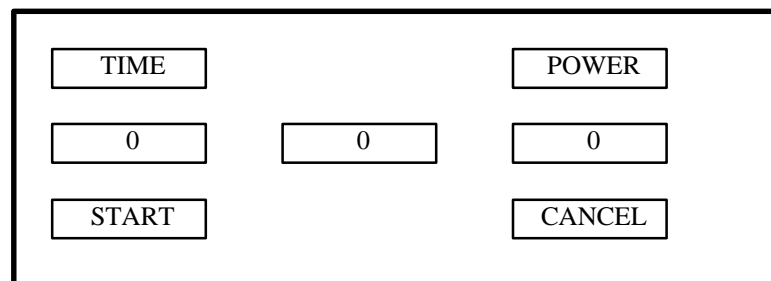


Figure 1.6. Microwave oven panel for World 3.

Scenario: Cooking porridge

Every morning, I eat a bowl of oatmeal porridge. (I don't get any commission for mentioning it in this book.) I cook it in my microwave oven according to the following procedure that I will call a 'conversation' to use a term used in the previous two examples:

Conversation:

1. I press the *Power* button, which can be interpreted as sending message 'Initiate cooking time setting function' to the oven.
 2. *Power* button message to oven electronics: 'Prepare to read length of cooking time'.
 3. Electronics to display: 'Show message for setting time'.
 4. Display performs the task and confirms: 'Done' to electronics.
 5. Electronics confirms to *Power* button: 'Done'.
 6. *Power* button to me: 'Message executed'. (I don't yet have a speaking oven and the button does not really tell me anything but the display indicates that my start-up 'message' has been completed.)
 7. I press the *1* button, effectively sending the message 'Set cooking time to 1 minute'.
 8. Button to electronics: 'Set time to 1 minute'.
 9. Electronics to itself: 'Set set cooking time to 1 minute'.
- and so on.

This example shows how the idea of objects and messages extends to inanimate objects. We find the idea of objects and messages is very natural and so did the designers of the first object-oriented language (Smalltalk). Any object-oriented programming language thus provides means for defining conceptualized objects, usually models of real world objects or concepts, and messages and for requesting operations and results from them. A pure object-oriented language such as Smalltalk does not provide any other means of control beyond objects and messages to create programs and it turns out that nothing else is needed.

Main lessons learned:

- Object-oriented problem solving means treating problems as mini worlds consisting of objects that understand messages and respond to them by performing tasks.
- By an object, we mean an equivalent of a black box that can respond to a predetermined set of messages. An object is often a simplified model of a real-world entity or concept.
- A message invokes an action that often generates a series of further messages to other objects.
- Every action in an object-oriented world is the result of sending a message to an object.
- A message has a sender and a receiver.
- The responsibilities of an object are the services that it is expected to be able to fulfill.
- In executing a message, an object may send a message to another object (its collaborator).
- Letting another object handle a part of a task is called delegation.
- When an object executes a message, it returns the result object to the sender of the message. This object may be just a confirmation that the message was completed.
- A scenario is a task encountered in a given problem area.
- A conversation is a sequence of message exchanges required to realize a scenario.

Exercises

1. Create an additional detailed scenario for each of the problem worlds presented in this section. Add new objects if necessary.
2. Complete the microwave conversation and draw its diagram.
3. Find objects needed to describe a basic version of the following problem worlds and list some of their responsibilities. Formulate at least one scenario for each problem world, give details of the corresponding conversation, and represent it graphically.
 - a. Checkout counter in a store.
 - b. Gas station.
 - c. Ice cream stand.
4. Execute the conversations presented in this section and the exercises above by assigning one person to each object. Each person-object gets an index card with the name of the assigned object on it. The holder of each card lists all the responsibilities of the assigned object on the card.

1.3 Examples of objects in computer applications

The situations described in the previous section illustrate the concept of objects and messages in real-world settings. We will now give several more examples to illustrate the object-oriented paradigm in computer settings.

World 4. A computerized library catalog

In this example, we will briefly shift our attention to the problem of *finding* the objects involved in a model rather than *using* them. We will not present any usage scenarios.

Problem: Our local library asked us to write a computer program implementing a simple library catalog. The program must keep track of borrowers (their names, addresses, telephone numbers, and borrowed books), and books (authors, title, publisher, library number, location, available or on loan). The program must allow the staff to add new borrowers, sign books out, search for books, and execute other scenarios.

Solution outline: To construct this program, we must first decide which objects are needed and what functionality they have. To do this, we formulate several scenarios and determine their actors, and combine this information with our general understanding of the problem. In this problem, we will limit our investigation to a simple analysis because the situation is familiar.

One of the obvious objects that we need is a computer representation of a *borrower*. The functionality of this object is essentially equivalent to the representation of a borrower that an old-fashioned library might store on catalog cards. For the purpose of the library, the borrower's functionality must include the ability to respond to messages such as 'Set name to ...', 'Set address to ...', 'What is your name?' and 'What is your borrower code?', 'Change your address to ...' and 'Change your borrower code to ...' corresponding to the ability of an index card to record and display information and to allow its change. In our context, we don't care what is the borrower's weight, color of hair, and place of birth although these characteristics are important features of physical borrowers and might be essential in another problem such as a police registry.

Another object required by our program is a model of a *book* and it includes those properties of a book that are relevant for a library catalog. We would expect the library catalog model of a book to include its author, title, library code, and on-loan information, but not the table of contents of the book, the text, and the pictures. The functionality of a model book object includes the ability to initialize and modify its data, and to answer messages such as 'What is your title?' and 'Are you in the library?', as well as 'Change your status to 'on loan' and so on.

We also need a *book catalog* object. This object is a collection of book information objects, being able to add and delete books, and search for books given the name of an author or title. We also we need a *catalog of borrowers* with similar functionality.

One notable feature of our solution is that although our computer models real-world objects, our model objects represent only their selected aspects. Computer model objects are not carbon copies of real-world objects.

World 5: Farm - a simulated animal world

This example revolves around several versions of our computer program called *Farm* that we designed to illustrate the main concepts of the object-oriented paradigm and to introduce Smalltalk programs. To use it, you must first open a *Farm Launcher* (Figure 1.7) by typing

Farm open

into a Workspace and executing this text with the *do it* command from the pop up menu.

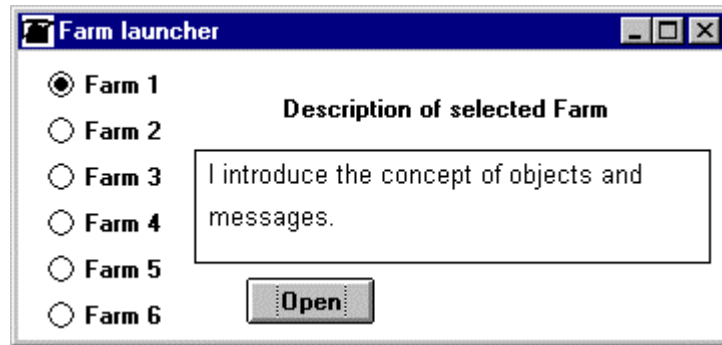


Figure 1.7. The opening window of *Farm Launcher*. Use it to select the first version of the *Farm* program.

As you see, the *Farm* program has six different versions and we will naturally start with *Farm 1*. When you select *Farm 1* as in Figure 1.7 and click the *Open* button, you will get the window in Figure 1.8 showing all animals living on this farm. As you can see, the farm is rather poor but we will show you in a moment how to add more animals.

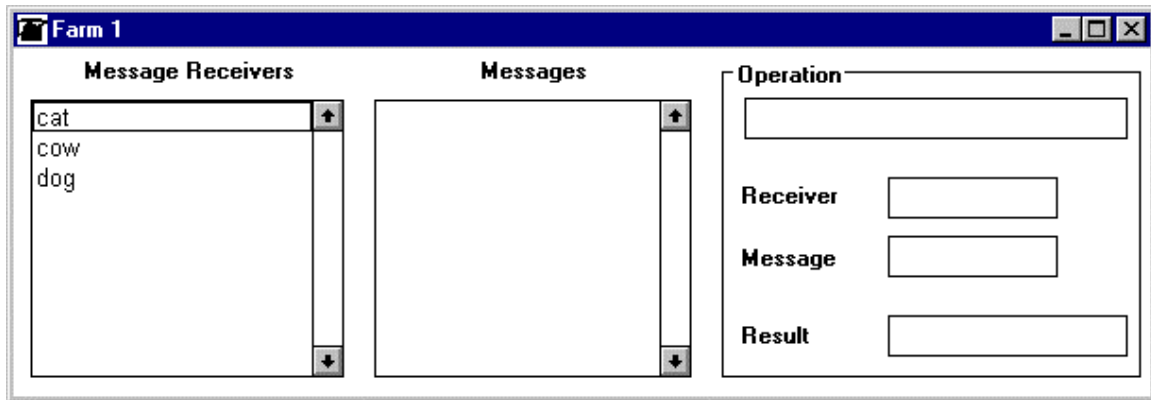


Figure 1.8. Initial state of Farm 1.

The animals on the farm are, of course, objects that understand messages such as run or moo. To execute a task, select an animal in the list on the left. This will display a list of the messages that the animal understands in the list in the middle as in Figure 1.9.

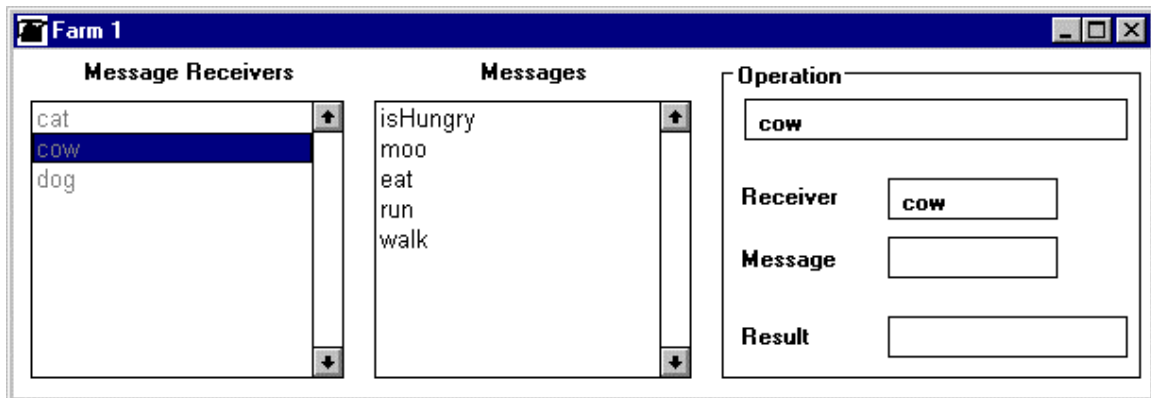


Figure 1.9. Farm 1 after selecting an animal.

When you now click a message, the program sends the message to the animal and shows a summary of the operation as in Figure 1.10 which shows that we executed expression

cow moo

whose receiver object is cow and whose message is moo, and that the result of the execution of the message –the cow’s response - is ‘I moo’.

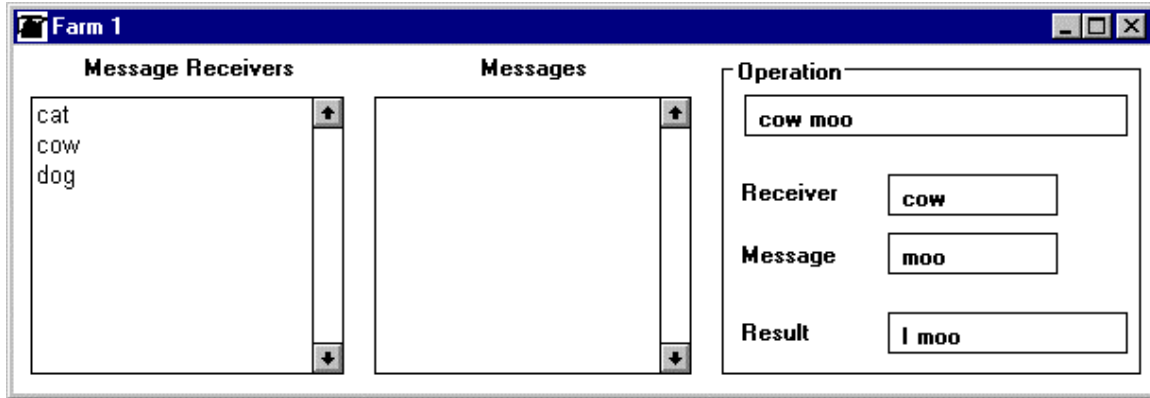


Figure 1.10. Result of sending message moo to object cow.

Farm 1 gave us a hands on experience with objects and showed that objects understand messages. Obviously, objects also have properties and *Farm 2* provides access to an extended set of messages that let you ask animals about their properties. Figure 1.11 shows you what these expanded message sets look like. A cow, for example, has a color and a name and it understands messages requesting these properties. We will leave it to you to try them out.

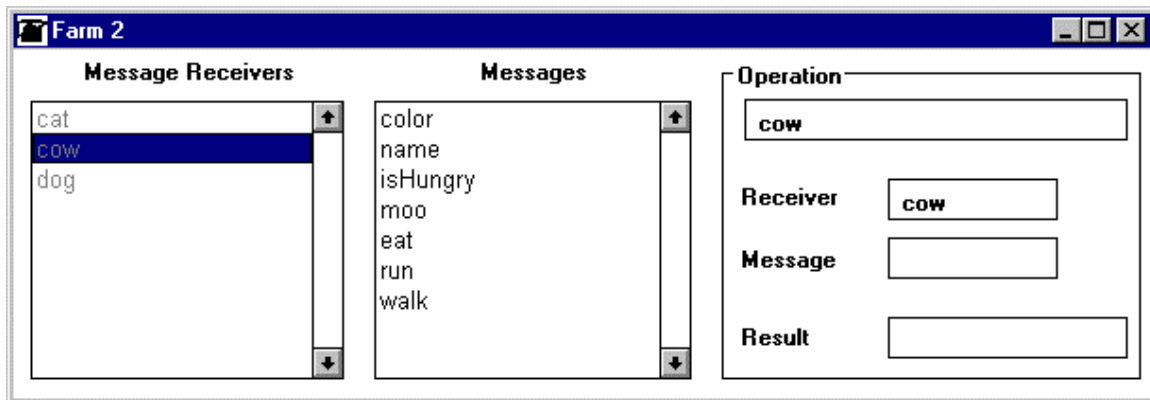


Figure 1.11. *Farm 2* animals have properties including name and color.

We promised that you will be to create new animals and *Farm 3* makes it possible. Its idea is somewhat different from *Farm 1* and *Farm 2* in that its window does not initially show any animals but rather ‘animal factories’ represented by the word Cat, Cow, and Dog (Figure 1.12).

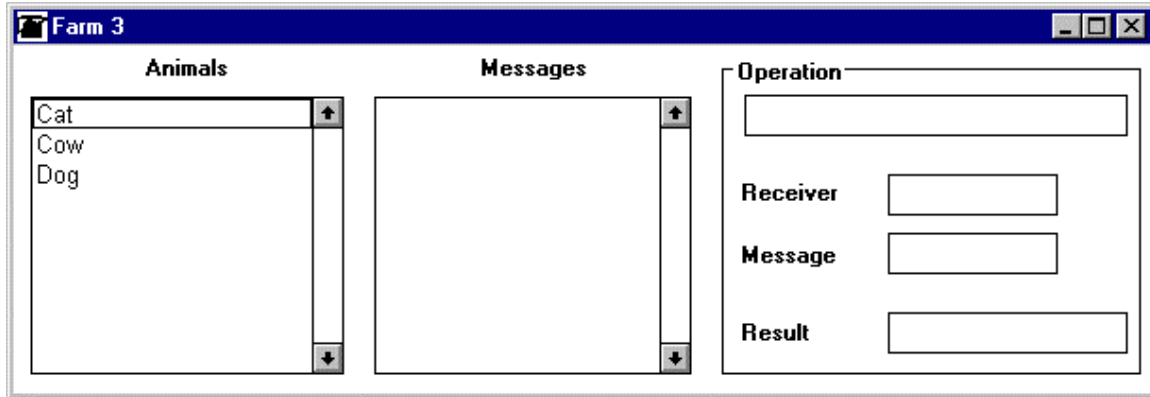


Figure 1.12. *Farm 3* adds animal factories.

Cow factories are, of course, quite different kinds of objects than cows. Cow factories don't understand messages such as `moo` or `eat` because they are not animals. Since animal factories can only make new animals, they only understand message `new` as you can see when you select Cow as in Figure 1.13.

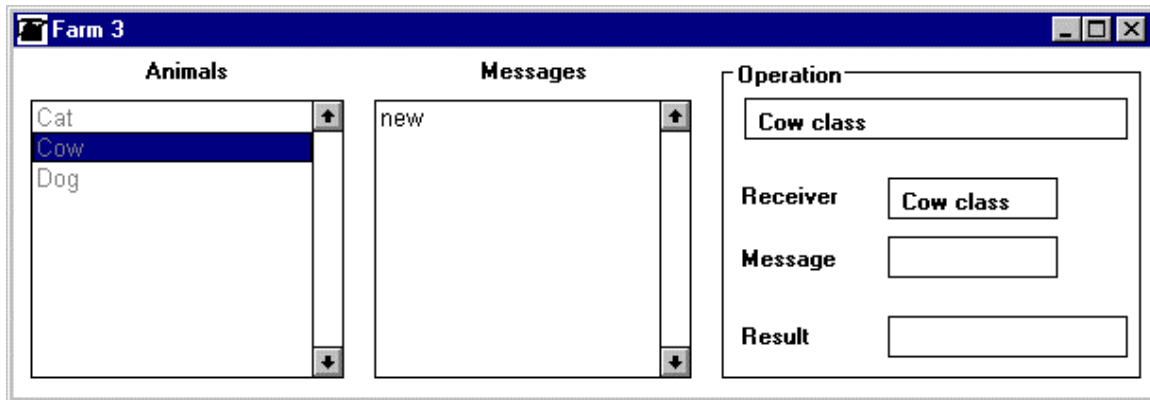


Figure 1.13. *Farm 3* introduces the concept of classes - object producing factories.

The distinction between objects and the factories that make them is natural. We are familiar with car factories (and they are a different kind of objects than cars), paint factories (and they are different from paints), and so on, and we can easily extrapolate our experience that objects must be produced to imagine that when a program wants to create for example a new window, it needs a window factory to do that, and that when it wants a rectangle, it must use a rectangle factory to create one. Object-oriented languages thus include object factories but instead of calling them factories, they call them *classes*⁴. We have shown this in our program which displays the name of the receiver as *Cow class*, for example, instead of *Cow factory* (right-hand side of Figure 1.13).

When you send the `new` message to an animal factory, the program requests information about the animal (its name and color) and then adds the animal to the farm (Figure 1.14). Since we can add as many animals as we want, the program gives each animal a number so that we can distinguish among them. In our example, we now have four objects: a Cat factory, a Cow factory, a Dog factory, and a cow displayed in the list as `cow1`⁵. The first three objects understand only message `new` (and create a corresponding animal),

⁴ We will see later that classes can serve other functions beyond creating objects, but creating objects is their most common function.

⁵ Following Smalltalk conventions, we distinguish instances from classes by deriving instance names from class names but starting class names such as *Cow* with a capital, and instance names such as *cow1* with a lower case letter.

the fourth is a cow with a name and a color and understands the same cow messages as in *Farm 2*. To distinguish the products from the factories, the products are called *instances* and our current farm thus has three classes (Cat, Cow, and Dog), and one instance of class Cow shown as cow1.

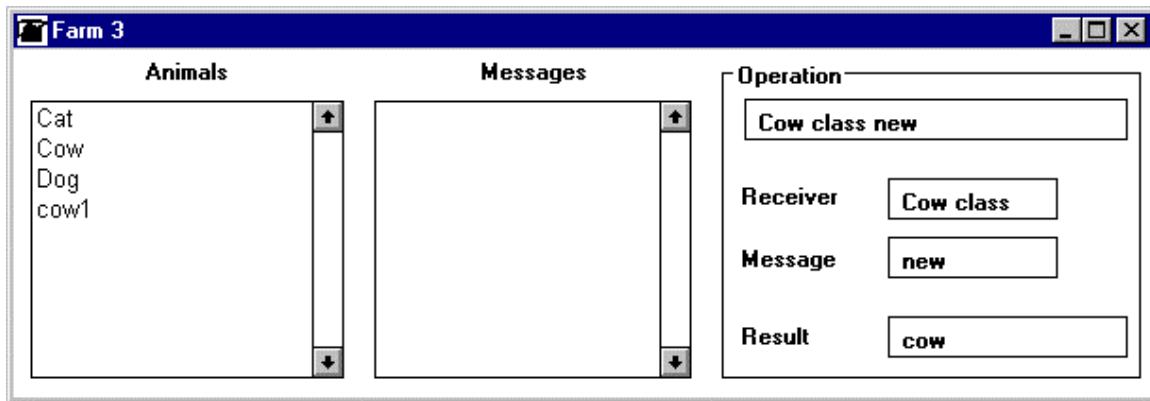


Figure 1.14. Farm 3 after creating a cow.

After playing with the farm and introducing the important concepts of class, instance, and object properties, let's now explore which other objects were needed to build the farm program itself.

The obvious objects required by *Farm* are classes Cat, Cow, and Dog. These objects are what is called *domain objects* and they describe the world that we are modeling – in this case a farm. But what about the instances of Cat and Cow and Dog, are they separate from the classes? The approach taken by Smalltalk is that a class is a blueprint for making instances and it thus knows how to make its instance. As a consequence, in addition to its own description a class also has a blueprint for creating its instances. As an example, class Cat has a blueprint for making cats and class Cow has a blueprint for making cows. Instance descriptions are thus always included in the definition of its class and we thus don't need separate objects for classes and their instances. We conclude that the domain objects for the Farm program are Cat, Cow, and Dog.

In addition to domain objects, we also need objects in charge of the user interface - objects that implement windows and their components such as buttons, labels, lists, and other 'widgets'. The window and the widgets must know how to display themselves on the screen and how to respond to user control such as clicking pressing a mouse button.

In addition to these rather obvious necessities, we also need a number of other objects to work in the background. As an example, we need an object "watching" the keyboard and the mouse, intercepting all keyboard and mouse events and sending them to our application. Since the screen may display any number of Smalltalk windows in addition to *Farm*, we also need an object that decides which window is currently active and should be responding to these 'input events'. And then we need mundane objects such as numbers, and strings of characters, and objects that hold collections of other objects, and so on.

The *Farm* is now beginning to look very complicated and if you looked behind the scenes and checked how many objects are active when *Farm* is running, you would find hundreds or thousands of them and probably around a hundred classes defining them! Does this mean that if we want to write a program such as *Farm*, we must invent hundreds of objects? Fortunately not. First, many of these objects are instances of the same class. As an example, the *Farm 3* window contains two list widgets (both instances of the same class), five labels (all instances of class Label), and so on. We conclude that although a running program requires many objects, the number of classes required to manufacture them is much smaller.

Moreover, we do not have to create all the classes that are involved in *Farm* because many other applications have similar needs and many generally useful objects such as buttons, windows, input sensors, and window schedulers have been created by the designers of Smalltalk and stored in its enormous *library* of prebuilt objects. To create *Farm*, we thus needed to design and implement only the specialized classes such as Cat, Cow, and Dog, and reuse those objects that are already in the library. We will, of course, store

our new objects in the library too and if we come across another application that needs them (such as an animated farm), we will reuse them as well. As we and other programmers develop new applications, the library will be getting more and more complete and if we designed our objects intelligently, fewer and fewer classes will have to be created from scratch.

Reuse of previously implemented objects is one of the greatest potential benefits of object-oriented environments such as Smalltalk. Reuse of existing blueprints is not, of course, an invention of object-oriented programming but an everyday fact of life. As an example, computers use power supplies, but manufacturers of computers don't have to draw blueprints of new power supplies because power supplies are also used in many other products and can be bought from a distributor. Similarly, if you need a screw, you don't need to invent one but rather buy it in a store and use it in whatever application you please.

In addition to minimizing the need to design new objects, reuse also contributes to quality. Commercially available components have presumably been carefully tested and are known to work well. VisualWorks Smalltalk takes the concept of reuse very seriously and its library contains more than 1,000 predefined and thoroughly tested reusable classes and many more can be bought from other software manufacturers.

A word of caution is in place. Although the experienced programmer gains much productivity from object libraries, a large library initially adds to the confusion of the beginner who finds the number of prebuilt components overwhelming. This is not a new problem either: The complete collection of all the tools of a professional carpenter is also enormous and incomprehensible to a handyman - but it is a prerequisite of high productivity and quality work for the professional.

World 6: Built-in Smalltalk objects

What kinds of objects would you expect to find in a programming environment? First of all, those that are needed to build the environment. Numbers, text, windows, buttons, labels, and pictures are some of the most obvious candidates and we will now briefly look at the most basic of them - numbers. This example is not a lesson on Smalltalk but only another illustration of the object-oriented paradigm.

Since computer hardware treats different kinds of numbers differently and since different kinds of numbers have different properties, most programming languages distinguish several kinds of numbers. In Smalltalk all numbers are, of course, objects and Smalltalk recognizes many kinds of numbers including the following ones:

- *Large and small integers.* Integers are numbers without a decimal point such as 15, -562, 0, and 4590100118912313108989887676465434111101. Smalltalk distinguishes small and large integers. 'Small' integers are those integers that your computer can handle directly, 'large' integers are those that Smalltalk must handle in a special way.
- *Floating-point numbers.* These are numbers with a decimal point, such as 156.43, -467.01, and 0.000012361 and computers handle them differently from integers. Smalltalk again distinguishes two kinds of floating-point numbers according to their range and number of valid digits (precision).
- *Fractions.* These are numbers represented by an integer numerator and an integer denominator such as 13/45 or -54/13. Computers don't have any instructions to deal with fractions and Smalltalk deals with them by using integer operations.
- *Complex numbers* are numbers with a real and an imaginary part. These objects are not part of the basic Smalltalk library and are included only in extensions; they are used mainly in engineering applications and they are not directly handled by hardware.
- *Fixed-point numbers* - similar to floating-point numbers but with a fixed number of digits behind the decimal point. Handled by Smalltalk, not directly by hardware.
- *Special numbers*, such as $+\infty$ and $-\infty$, more accurately the results of arithmetic operations that fall outside the legal range of floating-point values. These objects are included only in the extended Smalltalk library and some of them are handled by hardware.

If we want to treat numbers as objects, their functionality must allow us to ask number objects to do all the things that we normally do with numbers including arithmetic and calculation of mathematical functions. In this perspective, a calculation such as

3 + 5

must be treated as a message + to object 3 to do addition with object 5 as in

‘Object 3, do + with object 5 and return the resulting number object’

or, to put it differently, as

‘Object 3, tell me what is the result of doing + with object 5’

The functionality of number objects in Smalltalk thus includes the ability to respond to arithmetic messages such as + and -, the ability to calculate mathematical functions such as log or cos, the ability to compare themselves to other number objects (‘are you greater than 17?’), responding to questions such as ‘Are you an integer?’, and so on. This is, admittedly, an unusual but view of numbers but it is consistent with the OO view that everything is a number or a message.

Main lessons learned:

- Objects fall into two general categories – classes and instances.
- Classes can be thought of as factories and instances as their products.
- One of the main benefits of object orientation is its potential for reuse.
- Advantages of reuse include shorter application development time and better quality.

Exercises

1. Formulate a scenario for the library catalog world and expand it into a detailed conversation between the user and the program.
2. Experiment with the *Farm* program. As an example, explore how the combination of isHungry and eat messages works.
3. Another simulated microworld included in our software is a series of *Pen Worlds*. To open the program, execute PenWorld open using *do it*. Experiment with *Pen Worlds* in parallel with *Farm* worlds, using them to draw and erase colored lines and shapes on the screen. Formulate and solve tasks such as drawing overlapping and non-overlapping rectangles and other shapes in various colors. Pay attention to the object-message interface.
4. Formulate the following messages in terms of requests for information or action:
3 squared
pi convertedToDegrees
3 / 5(creates a fraction)
pi cos (calculates the cosine of π)

1.4 How does an object-oriented application work?

Although you are now comfortable with the principle of object-oriented software, you may be wondering how object-oriented programs work. Your question might be: ‘OK, so we have a pile of objects stored in the computer, but how do they come to life and cooperate to solve a problem?’ To answer this question, the following greatly simplified script describes the operation of the *Farm* program. We present the script in an anthropomorphic form, treating the objects as if they were living organisms or robots, a perspective that many people find useful when thinking about objects.

To start the *Farm*, the user enters and highlights the expression

Farm open

and executes it with the *do it* command. This sends the *open* message to *Farm* and Smalltalk starts by looking for the definition of the *open* message in *Farm*. It finds the definition and discovers that its operation starts by a request to the *Window* class object to create an instance of itself from a specification provided by the *Farm* class, and draw it and all its components on the screen. In executing this request, the window object constructs a detailed description of the frame of the window with a label and sends a message to each of its button, label, and other 'widget' objects, asking them to draw their representation and to get ready to respond to mouse clicks. When this is done, the window asks a mouse/keyboard input sensor object to start tracking the user's actions.

The input sensor object starts monitoring keyboard and mouse buttons and when the user clicks the mouse over, for example, the *Open* button, the input sensor passes this information to the button object, which then executes the function for which it is programmed. In this case, the message is to open a window for *Farm 1*. When the message is completely executed and the *Farm 1* window opens on the screen, the input sensor starts tracking the mouse again, waiting for another mouse click, and so on, until the user closes the *Farm 1* window and the *Farm* launcher and terminates the application.

We conclude that a typical application operates by a combination of user input events that trigger messages to objects which then create a chain of message interchanges between objects.

Main lessons learned:

- To start the execution of an object-oriented program, send a start-up message to an object designated by the application as the start-up object. This begins a sequence of message sends, usually involving interaction with the user.

1.5 Classes and their instances

If you wondered whether the concept of a class violates our claim that the Smalltalk environment is totally populated by objects, we can reassure you. A class is simply a special kind of object whose main role usually is to manufacture instances. The following table illustrates these notions and contrasts classes and their instances on the example of animals and animal factories:

class (producer)	its instance (product)
Cow (can create new cows)	cow1 (can moo, eat, etc.)
Cat (can create new cats)	cat3 (can meow, eat, etc.)

Although classes and their instances are related, they are quite different kinds of objects. In particular, classes understand one set of messages, and their instances understand a different set of messages. As an example, *class Cow* understands the *new* message that creates a cow, but it does not understand and cannot execute messages such as *moo*, *eat*, or *run*. *Instances* of class *Cow*, on the other hand, understand *moo*, *eat*, and *run* but don't understand *new*.

To avoid confusion between messages understood by classes and messages understood by their instances, Smalltalk distinguishes *class messages* (those understood by a class), and *instance messages* (those understood by instances). For example, message *new* is a class message, but message *moo* is an instance message. The following table contains a summary of class and instance messages for class *Cow*:

Cow <i>class</i> messages	Cow <i>instance</i> messages
---------------------------	------------------------------

new	color name isHungry moo eat run walk
-----	--

We have already noted that the class definition defines the blueprint of its instances, and both class messages and instance messages are thus a part of the definition of the class. Class messages are on the class side of the definition, instance messages are on the instance side.

As we have seen, if we want to create a new instance of a class we send a class message to the class asking it to create an instance. But where do classes come from? When Smalltalk programmers need a new kind of object, they must write a definition with detailed descriptions of the class and instance messages and add it to the Smalltalk library of classes. To do this, they use a special tool called the Browser. We will see what the Browser looks like in the next section.

Since Smalltalk is an object-oriented language, Smalltalk programming consists mainly of creating new classes and reusing or expanding the ones that already exist. The advantages of Smalltalk over other object-oriented environments include that Smalltalk makes creation of new class definitions very easy, that it contains many predefined classes, and that both built-in and new classes can be reused and modified with equal ease. This greatly increases programming productivity and improves program quality because large parts of new applications consist of previously created and exhaustively tested classes.

Main lesson learned:

- Object-oriented environments distinguish two kinds of objects - classes and instances.
- Classes are used mainly to create instances, instances are the workhorses which perform most of the work.
- The messages understood by a class are distinct from the messages understood by its instances. Messages understood by a class are called class messages, messages understood by its instances are called instance messages.
- The definition of a class contains the definition of both its class messages and its instance messages.

Exercises

1. As we mentioned, class messages are used mainly to create new instances. There are, however, many class methods that perform other functions such as initialization, opening of applications (as in Farm open), provide useful information and frequently needed operations, demonstrate how to use a class, and so on. As an example, execute the following class messages in the Workspace with *print it*:
 - a. Time totalSeconds "Answer the total seconds since 1901 began"
 - b. Date nameOfDay: 3 "Answer third day of the week."
 - c. Float pi "Answer value of π ."
 - d. ScheduledWindow openNewIn: (100@100 corner: 200@300) "Open window with upper left corner at point (100, 100) and lower right corner at point (200, 300)."
 - e. Float radiansPerDegree "Answer number of radians in one degree."
 - f. Window platformName "Return the name of the operating system."
 - g. Dialog warn: 'Click OK to continue' "Open a notification window."
 - h. Rectangle fromUser "Get a rectangle from the user."
 - i. GraphicsContext exampleArcs "Execute example program."

1.6. A first look at Smalltalk classes

To view classes in the Smalltalk library and to add, delete, or modify classes, Smalltalk programmers use the *System Browser*. To open it, use the *Browse* command in the *Visual Launcher* or simply click the *Browser* button in the launcher window (Figure 1.15).

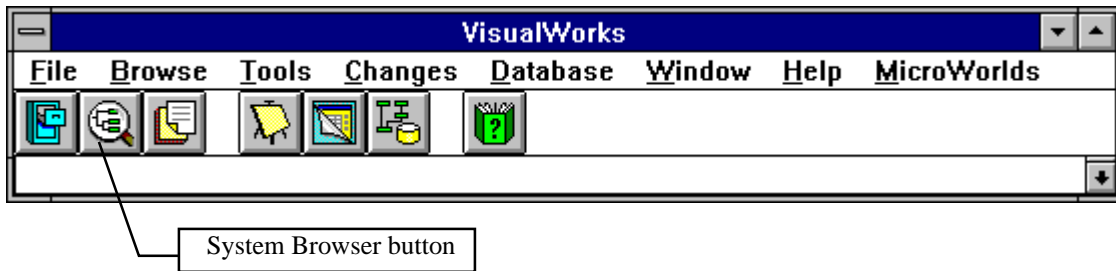


Figure 1.15. To open the System Browser, click the *Browser* button or the *Browse* command.

When the System Browser opens, it looks as in Figure 1.16. It contains four 'views' at the top, a text area at the bottom, and two buttons labeled *instance* and *class*; these buttons are used for selecting between class and instance methods. The leftmost top view is called the *category view* because it contains a list of *categories* of all classes in the class library. Each category contains several related classes and each class is included in exactly one category. The purpose of categories is strictly organizational - they group together related classes to make it easier to find a class in the enormous library. Categories have nothing to do with class behavior.

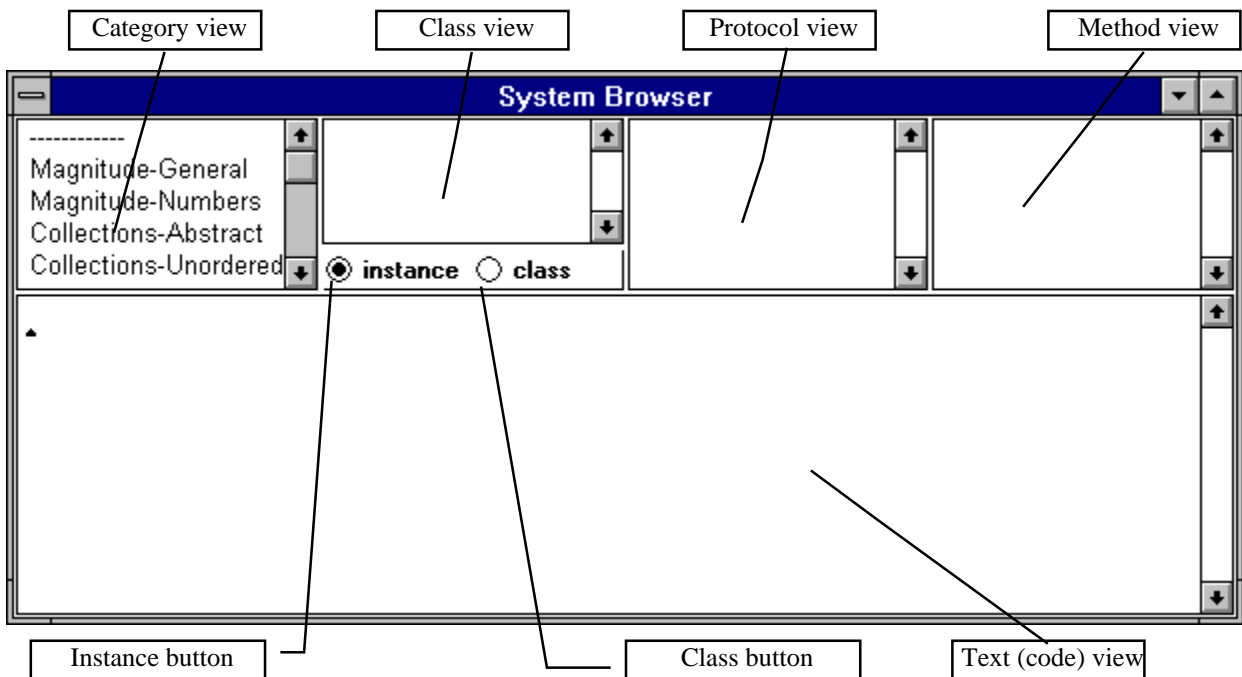


Figure 1.16. A new System Browser and its parts.

To access the definition of a class or to create a new one, click the name of its category. If you don't know which category contains the desired class, use 'find class' in the <operate> menu of the category view. When you select a category, the *class view* displays all classes in this category (Figure 1.17) and the text view at the bottom of the window displays a Smalltalk expression that can be used to add new classes; this part of the Browser is irrelevant at this point. To examine information about a class, select the class in the *class view*. In our example, we selected category *Magnitude-Numbers*, scrolled the class view down, and

selected class Fraction. The text view at the bottom of the Browser now shows the Smalltalk messages that created the class.

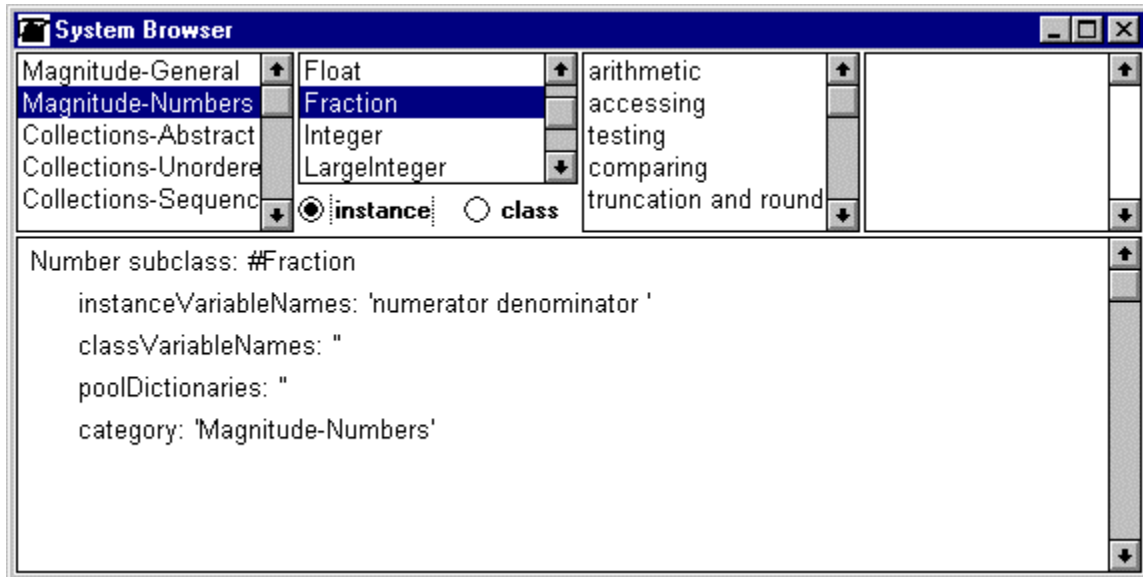


Figure 1.17. System Browser showing the definition of class Fraction. Details will be explained later.

The text view now displays the definition of the class and provides access to definitions of all messages defined in the class. This information can be accessed from popup menus or by making further selections in the remaining views of the browser. In this section, we are not yet interested in Smalltalk code but we might want to look at the comment that programmers write for a new class to describe its purpose and structure. To view the comment, open the <operate> menu in the class view (Figure 1.18), select the *comment* command and the comment of the currently selected class will appear in the browser.

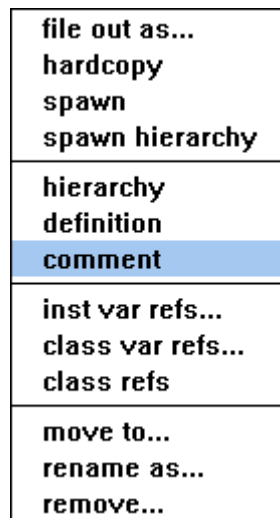


Figure 1.18. To display the comment of a selected class use the <operate> menu.

You will have to learn how to use the System Browser because it is the most important part of Smalltalk's development environment.

Main lessons learned:

- To access a class in the Smalltalk library, use the System Browser.
- The System Browser lets you view and edit definitions of all classes in the library and add new ones.
- The System Browser groups classes into categories. Each class is in one category and categories don't overlap.
- The purpose of a category is purely organizational and has no effect on the behavior of the class.
- The System Browser is the most important Smalltalk tool.

Exercises

1. Use the System Browser to display the comments of the following classes and print them out using command *hardcopy* in the text view's <operate> menu.
 - a. Date (category Magnitude-General)
 - b. Fraction (category Magnitude-Numbers)
2. List all classes in category Collections-Abstract.
3. Instead of opening the System Browser and viewing the whole library, you can open a specialized browser on a selected category, a selected class, or even a smaller part of the library. To do this, select the category or class in the browser and use the *spawn* command. Use this technique to open a category browser on category Magnitude-Numbers, and a class browser on class Fraction.

1.7 Object properties

We have already seen that objects understand messages and usually contain information. As an example, an instance of Cow knows its name and color and knows whether it is hungry or not. Those properties of an object that can change during its lifetime are often referred to as its *state* properties.

If we are working with several cows, each cow may be in a different state at any given moment and each must therefore carry its state information with it. References to an object's properties are called *instance variables* because the value of the state object to which they refer may change during the object's lifetime. As an example a cow that has just eaten will not be hungry for a while - but eventually will get hungry again and will need to eat. The state of the object referred to by the variable that keeps track of this aspect of the cow's condition will thus vary between true and false as the state of the cow object changes. Each instance variable of a given object has its own distinct name so that we can refer to it when we need to access or manipulate it via the object's functionality.

We can thus think of an object (Figure 1.19) as capsule consisting of a state (held in instance variables) and functionality (defined by *methods* - detailed definitions of the behavior of each individual *message*). When an object receives a message, Smalltalk looks up the corresponding method in the receiver's class and executes its definition; in this process, the method may change the object's state and thus the values of its instance variables. A list of all variables that describe an object's state are a part of its blueprint and are kept by the class of the object. You can find them in the System Browser.

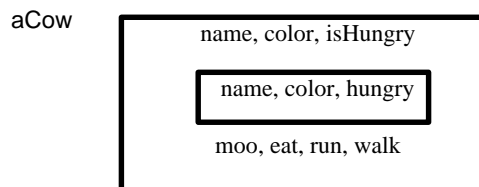


Figure 1.19. An object encapsulates state and provides functionality via messages.

As an illustration of this principle, the definition of class Cow contains

- definition of method *new* for creating new cow instances (*class* method of Cow),

- definitions of methods name, moo, eat, and so on (*instance* methods of Cow),
- list of names of instance variables holding a cow's name, color, and isHungry.

While the class has a list of instance variable names, each instance of Cow holds the *values* of its instance variables, and is aware of its class so that when it gets a message, it can access the corresponding method describing how to execute the message (Figure 1.20).

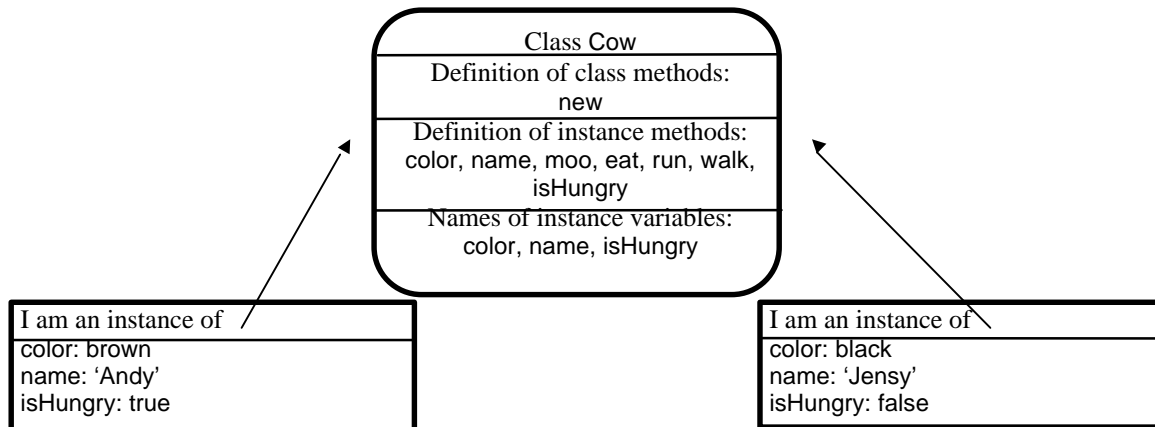


Figure 1.20. Each object knows its class and the values of its instance variables. The definition of a class contains the instance blueprint: a list of instance variable names and definitions of class and instance methods.

Our existing model of a class definition is not symmetric because it has no counterpart of instance variables. In reality, many classes also have *class variables* because not only instances but even classes themselves may have properties. Class variables are typically used for one of three reasons. One is that a class may need to keep some generally useful information, another is that a class may need to keep some information needed to create its instances, and the third is that it may need to know something common to all instances.

As an example of a class variable holding *generally useful information*, consider that many numeric applications need to know the value of π . Since $\pi = 3.14159\dots$ - a floating-point number - one would expect that floating-point numbers should be able to supply this information. It would be wasteful for each floating-point number to carry along an identical copy of the same π object, and the value of π is thus stored only once - as the value of *class variable* Pi declared in class Float.

As an example of information that a class needs *to construct its instances*, consider text objects. All text objects are normally displayed with the same font and the class that produces text objects needs to know what this default font is. The default font could thus be stored in a class variable. (The reality is somewhat more complicated.)

An example of information that a class might need to know about its instances is found in class Window: It is often useful to know which window on the screen is currently active and class Window keeps this information in class variable CurrentWindow.

We conclude that a more realistic picture of the general structure of a class definition is as in Figure 1.21.

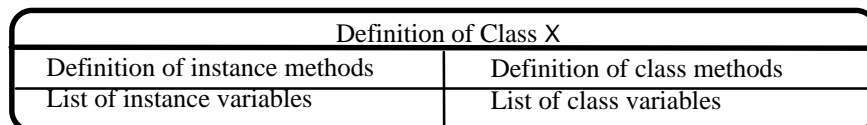


Figure 1.21. A class definition may include class and instance methods, and class and instance variables.

Values of instance and class variables are objects

After introducing the concept of variables as holders of object properties, it is now time to examine the nature of these properties. As one could expect, *values of instance and class variables are again objects*. This principle may extend to a great depth because these objects often have their own properties which are again objects, and the structure of an object may continue branching for a long time. Many objects are thus nested assemblies of objects and can be represented as graphs of chained references from one object to another (Figure 1.22). Eventually, of course, the chain of references must stop and some objects must contain 'real' values rather than just references to other objects. These elementary objects represent things such as basic kinds of numbers, and characters (letters, punctuation, and digits).

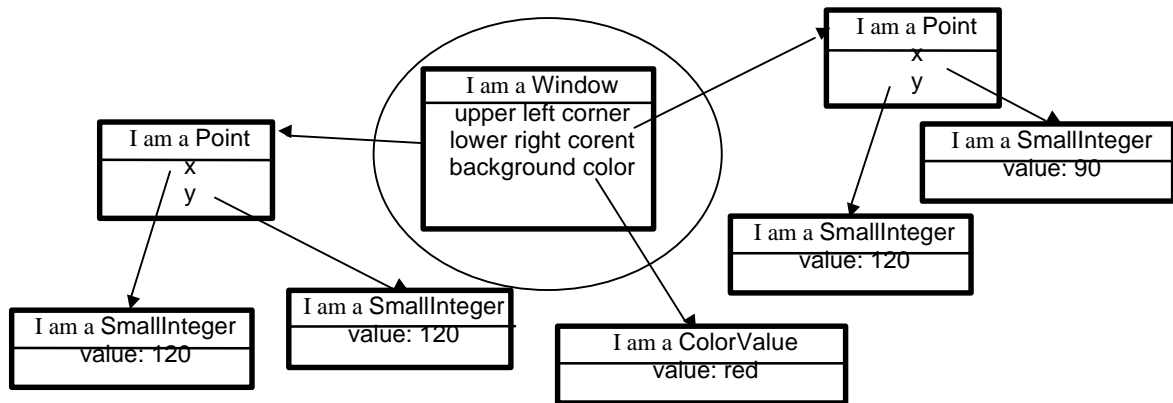


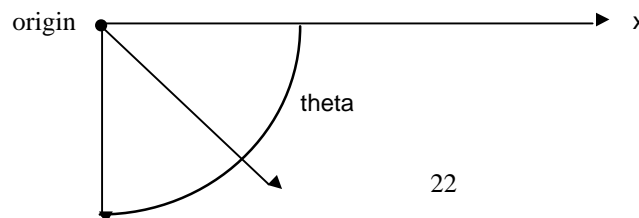
Figure 1.22. Values of instance and class variables are objects. (An idealized representation.)

The nested nature of Smalltalk objects is analogous to the nested structure of objects in the real world. As an example, a computer consists of a chassis, a motherboard, a power supply, adapter cards, connectors, and other components. A motherboard, in turn, consists of a printed circuit board, chips, connections, and connectors. A chip consists of a package, pins, an electronic circuit, and so on. Eventually, further decomposition becomes meaningless and we reach 'elementary' objects. We conclude that although the structure of objects may be very complicated, the fundamental simple principle remains unchanged - everything is an object.

Information hiding

An important property of Smalltalk objects is that their state components are hidden from other objects. An object is thus like a black box which contains some functionality and has an internal state, but the values of its components cannot be seen or changed from the outside, unless the object understands messages that access these values. In fact, other objects cannot even know what the internal representation of an object is. This property is called *information hiding* and it means that the only way to get at an object's state is via the messages that it understands.

As an example of information hiding, class Point which is a blueprint for points on the screen includes methods called x and y which return the point's x and y 'Cartesian' coordinates, and messages r and theta which return its 'polar' coordinates (Figure 1.23). Since Cartesian and polar representations can be converted into one another, we could implement Point using either instance variables x and y, or instance variables r and theta. If we used the polar implementation, methods r and theta would simply return the values of the corresponding instance variables whereas methods x and y would have to calculate x and y from the polar coordinates. In reality, Point is implemented with Cartesian representation and methods x and y return the values of instance variables x and y without calculating anything, and r and theta calculate their values from x and y. This fact is, however, hidden from other objects which can only question points using messages x, y, r, and theta.



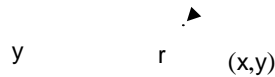


Figure 1.23. A point can be represented by Cartesian coordinates x and y or by polar coordinates r and theta.

Information hiding provides great freedom to implementers. As an example, if we decided to change Point to use polar coordinates, all programs using points would still work provided that we change the implementation of all Point methods to reflect the new implementation, because other objects can only use points through their *message interface* - by sending messages. Information hiding thus means that properly implemented changes to internal representation do not affect other objects.

To appreciate the advantage of information hiding, assume that the designer of a class such as Window could access instance variables of another class such as Point directly, without using messages, and use the values of x and y for some calculations. If somebody changed the implementation of Point to use Cartesian coordinates x and y instead of polar coordinates r and theta, Window would no longer work because x and y would no longer exist. And if window stopped working, the browser would not work and neither would any applications using window interfaces. Because of information hiding, class Window communicates with points only through their message interface and Window will thus work even if Point changes as long as the message interface of Point is properly maintained.

Before closing this section, it is worth noting that objects don't have to have instance or class variables. As an example, instances of class True don't have instance variables because they don't need any. Although the Smalltalk library includes many classes with no instance and class variables, all Smalltalk objects have functionality. There are no objects that don't understand any messages because such objects would be totally useless.

Main lessons learned:

- Objects encapsulate functionality and state.
- Each object carries values of its state properties with it.
- References to state properties of classes are called class variables, references to state properties of instances are called instance variables.
- Values of class and instance variables are objects.
- The definition of a message is called a method.
- Method definitions and names of class and instance variables are included in the definition of the class.
- Values of state properties are hidden inside objects and other objects can access them only via accessing methods – if the class definition provides them.
- The collection of all messages understood by an object is called its message interface.
- Communication with an object is only through its message interface.

Exercises

1. Explore the properties of pens and erasers using the *Pen World*.
2. List the major components of the following objects and expand the internal structure of some of the components to a reasonable number of levels.
 - a. A car from the perspective of a car mechanic.
 - b. A car from the perspective of a car salesman.
 - c. A bank from the perspective of a bank teller.
 - d. A bank from the perspective of a bank customer.
 - e. An art museum from the point of view of an art historian.
 - f. An art museum from the perspective of a civil engineer.
3. An object with instance variables but no methods holds information so why would it be useless?

1.8 Using System Browser to find out about objects

To view or edit the state variables and methods of a class, open the System Browser and select the desired class. The text view will display the definition of the class, and the commands in the <operate> menu of the class view allow you to display the comment and obtain all sorts of other information about the class.

Depending on the selection of the *instance/class* pair of buttons, the protocol view shows all instance or class *protocols* of the class. The function of protocols is similar to that of categories - to organize methods for easier access; and just like categories, protocols don't have any effect on object behavior. If you want to view a method, select its protocol (if you know which one it is) and click the methods name, or use the *methods* command in the protocols' <operate> menu. In our example, we selected class Fraction and instance protocol arithmetic and obtained the display in Figure 1.24.

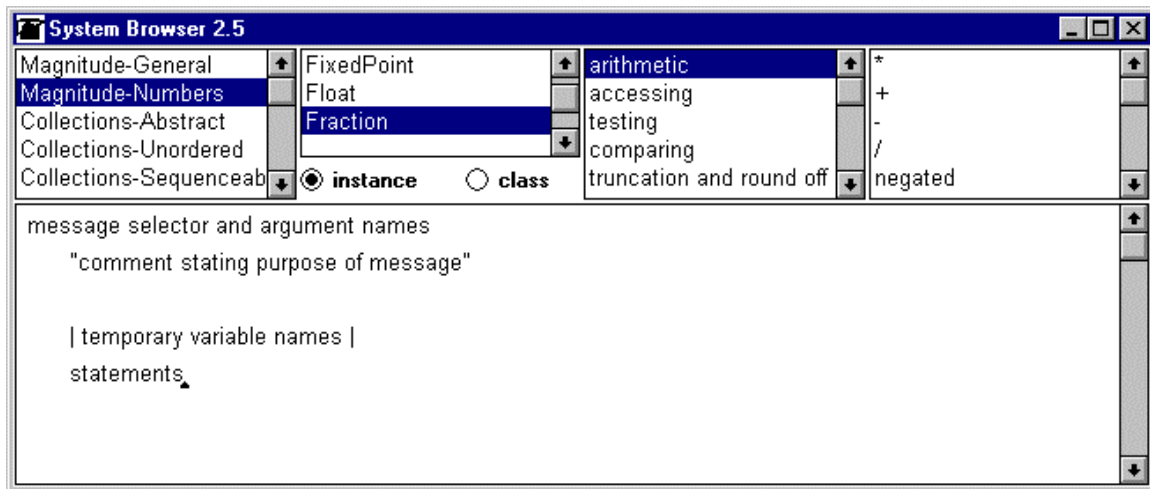


Figure 1.24. Methods in *instance* protocol arithmetic in class Fraction.

The method view on the right shows all methods in the selected protocol and the text view shows a template that can be used to create a new method. When you select a method in the method view, its Smalltalk code appears in the text view as in Figure 1.25 and you can now read it, find all references to it, modify it, delete it, and so on. We will learn later how to do this and how to understand the code.

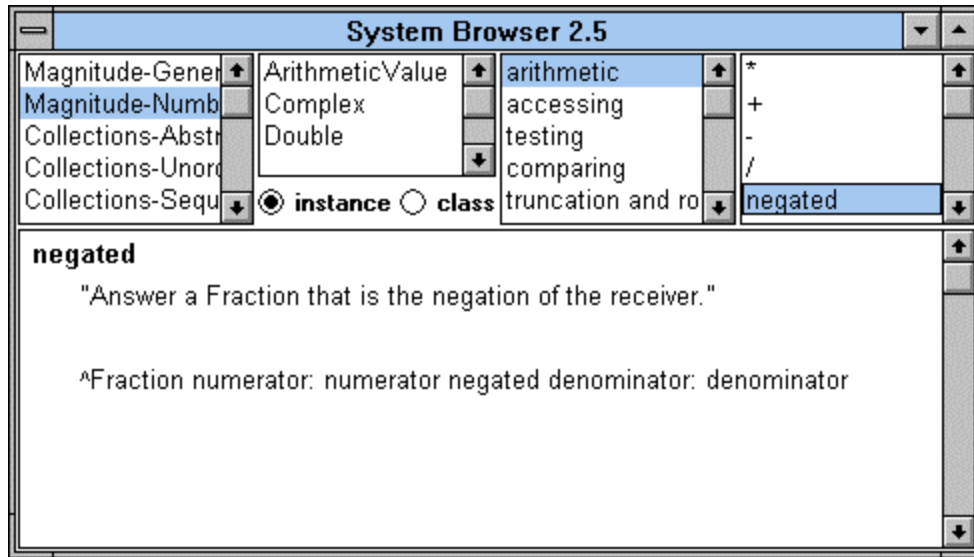


Figure 1.25. Definition of instance method `negated` in the arithmetic protocol of class `Fraction`.

Main lessons learned:

- Class methods and instance methods are divided into protocols.
- Each method is in one protocol and protocols don't overlap.
- The concept of a protocol is purely organizational and has no impact on behavior.

Exercises

- List all instance variables of the following classes:
 - Circle
 - LineSegment
- List all instance methods of `Fraction` that use instance variable `numerator`. (Hint: Use command *inst var refs* in the <operate> menu of the class view of `Fraction`.)
- List all references to class `Date`. (Hint: Use <operate> menu command *class refs*.)
- List all class variables defined in the following classes:
 - Float (category Magnitude-Numbers)
 - Date
- List all instance protocols defined in the following classes:
 - Character (category Magnitude-General)
 - Date
 - Number (category Magnitude-Numbers)
- Repeat the previous exercise for class protocols.
- List all methods in the following classes and protocols:
 - Class `Fraction`, instance protocol `arithmetic`
 - Class `Date`, instance protocol `inquiries`
 - Class `Date`, class protocol `creation`
- List all methods that send message `today` defined in class `Date`. (There are two ways to do this - either by selecting the method in the browser and executing *senders* from the method view's <operate> menu, or by using command *References To...* in the Visual Launcher (Figure 1.26).
- Find all classes that define method `+`. (Either open a browser on one such definition - for example in `Fraction` - and execute command *implementers* from the method view's <operate> menu, or use command *Implementers* in the Visual Launcher).
- Find all messages sent by class method `addTime:` in class `Time`. (Hint: Select the method in the browser and execute command *messages* in the <operate> menu of the method view.)

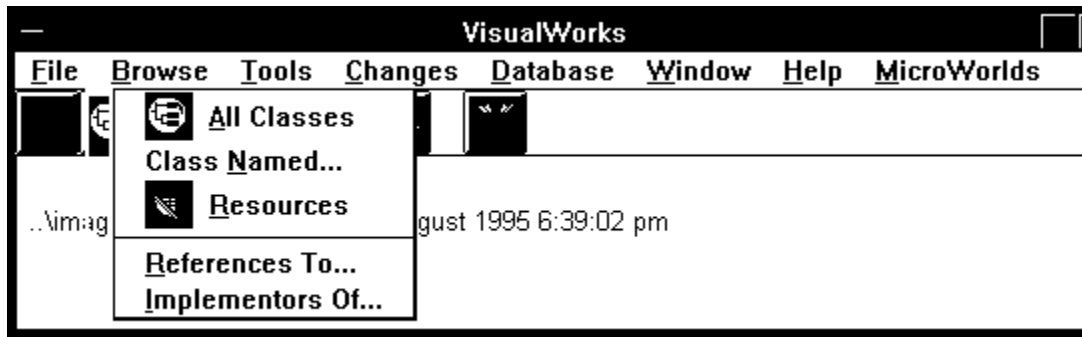


Figure 1.26. To view implementors or references to a method, use the *Browse* drop down menu.

1.9 Class, subclass, superclass, abstract class, inheritance, class hierarchy

Up to now, we dealt with classes in isolation. In this section, we will consider classes in relation to one another in the way that biologists classify plants and animals.

The system for the classification of animals looks like an upside-down tree (Figure 1.27) in which items at the bottom are animals such as lions or tigers, and higher up items are abstractions - such as mammals. Of course, even the lion at the bottom of the tree is not a real lion but a concept that defines what a lion is. In our terminology, this concept corresponds to a class. A particular lion in the wild or in a ZOO, on the other hand, is an instance of this class.

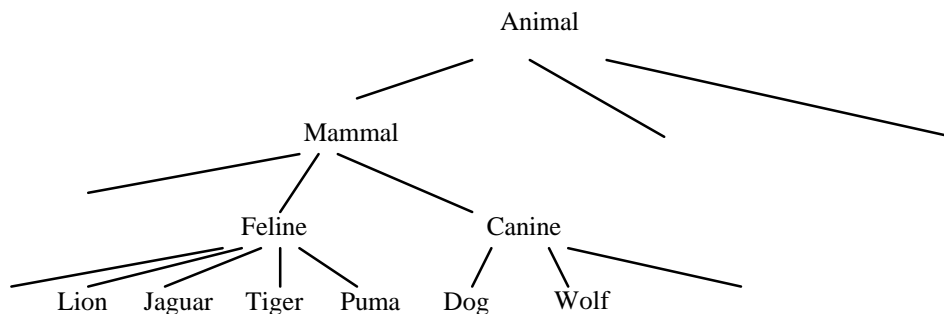


Figure 1.27. The classification tree of animals.

In our diagram, a Lion is a special kind of Feline. Similarly, a Tiger is a special kind of Feline, and so is a Puma, and a Jaguar. In class terminology, we would say that class Lion is a special kind of class Feline - a *subclass* of class Feline. Inversely, class Feline is a generalization of classes Lion, Jaguar, Tiger, and Puma - a *superclass* of Lion, Jaguar, Tiger, and Puma. The classification scheme thus defines a *hierarchy of classes, subclasses, and superclasses* and the arrangement reflects generalization (going up) and specialization (going down).

The purpose of a class hierarchy is to *factor out* shared features of related classes. As an example, since Lion is a special kind of Feline, it has all the properties and behaviors of Feline - and several additional specialized ones. Similarly a Tiger - being another subclass of Feline - has all the properties of Feline, and some additional ones. By constructing a hierarchy, we can thus gather all properties common to several related classes in one place - the superclass - and pass them down to all subclasses implicitly, without having to repeat them for each subclass. A description of a Tiger, for example, may then be quite short because we can say that a Tiger is a Feline and has certain special additional properties listed in the description. This passing of superclass properties and behaviors down to subclasses is called *inheritance*.

Besides making specifications shorter, inheritance has several other benefits. One is that by listing shared properties and behaviors in one place, we can avoid omissions and mistakes that we might commit if we kept a complete detailed description for each animal separately. Keeping shared properties in one place

also means that if we decide to modify them because we have a new insight into the nature of the subclasses or because we made a mistake, we only need to modify the superclass. This again saves work and prevents omissions such as modifying individually all subclasses of *Feline* but forgetting to modify the *Lion* class.

Our example shows that class hierarchies may be any number of levels deep. *Feline* is a subclass of *Mammal*, together with *Canine*. Class *Feline* thus has all properties of *Mammal* and since *Lion* has all the properties of *Feline*, *Lion* also has all the properties of *Mammal*. In other words, a class inherits not only the properties of its immediate superclass, but the properties of *all* its superclasses up to the top of the hierarchy tree. This increases the savings and the security obtained by superclassing - but makes deeper hierarchies more difficult to understand.

Another look at Figure 1.27 suggests another important concept. The classes at the junctions (nodes) of the tree can be divided into two groups: those that define real animals (such as *Lion*, *Jaguar*, and *Puma*), and those that are only abstractions (such as *Feline* and *Mammal*). Classes that correspond to real animals have instances in the real world while those that represent abstractions don't: There is no physical animal derived directly from the abstract concept *Feline*, there is only a physical *Lion*, *Puma*, or *Jaguar*. In OO terminology, classes representing pure abstractions are called *abstract classes* whereas those that can be instantiated (represent real objects) are called *concrete classes*. The essence of abstract classes is that they are not used to create instances and their only purpose is to factor out shared behavior and state information and pass it down to the concrete subclasses.

As we have seen, the main point of subclassing is to take advantage of inheritance, and object-oriented languages distinguish two kinds of inheritance - single, and multiple. The kind of inheritance that we used so far is called *single inheritance* because it limits the number of direct superclasses that a class may have to one - each class has a single superclass (the class at the top of the tree of course does not have any superclasses). In *multiple inheritance*, a class may have any number of immediate superclasses. While the animal kingdom tree is an example of single inheritance, a real world parallel of multiple inheritance is the human family: Every human being has two biological parents and inherits some properties from one and other properties from the other. Single and multiple inheritance correspond to class hierarchy structures depicted in Figure 1.28.

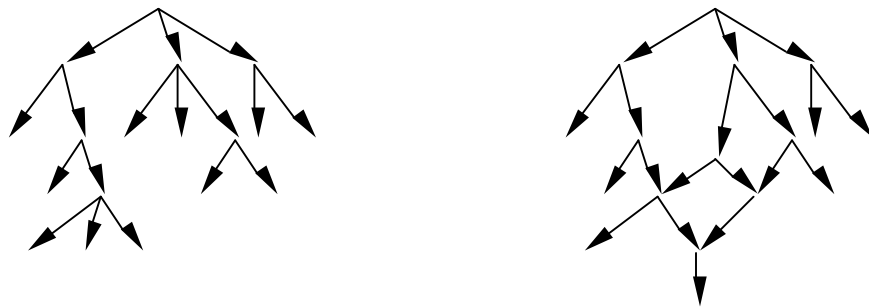


Figure 1.28. Single inheritance (left) and multiple inheritance (right).

The debate over which kind of inheritance is better is unresolved because each has its advantages and disadvantages. The advantage of multiple inheritance is that it is sometimes very natural. As an example, data files can be divided into read-only files, write-only files, and read-write files. If we were to design a class hierarchy for files, we would probably want to define one class for read-only files, one class for write-only files, and define the class for read-write files as a subclass of both (Figure 1.29). Class read-write file is thus a natural candidate for multiple inheritance. If the language does not support multiple inheritance, we must make class read-write file a subclass of read-only file, for example, and copy all the writing properties from the write-only file class. This is an unpleasant duplication.



ReadWriteFile

ReadWriteFile

Figure 1.29. Class ReadWriteFile in a language with multiple inheritance (left) can inherit its functionality from two superclasses. In a language with single inheritance (right) some functionality must be duplicated.

The disadvantage of multiple inheritance is that the exact nature of inheritance in non-trivial hierarchies may be very difficult to understand, especially if some superclasses share identically named but different methods or variables. Moreover, control of the effect of changes in higher level classes on classes at the bottom of the hierarchy tree can be disastrous. This is, of course, a problem with single inheritance as well but it becomes more difficult with multiple inheritance.

Commercial implementations of Smalltalk take the view that ease of understanding and maintenance are more important and use single inheritance. (There is also the historic legacy that the original Smalltalk used single inheritance.) There is, however, no built-in restriction against multiple inheritance and multiple inheritance extensions of Smalltalk has also been implemented. Some other programming languages such as C++ allow multiple inheritance but Java uses single inheritance.

Main lesson learned:

- Classes are organized in a tree-like hierarchy, leading to the concepts of a superclass, subclass, and inheritance.
- Going up the hierarchy tree corresponds to generalization, going down corresponds to specialization.
- The main benefit of subclassing is inheritance: A subclass inherits all properties of its superclass.
- Subclassing should be used when the new class adds new behaviors to an existing class. In other words, inheritance should be additive.
- Since each class inherits from its superclass, each class inherits from all its superclasses. Technically speaking, inheritance is transitive.
- There are two types of inheritance - single and multiple. In single inheritance, a class may only have one immediate superclass. In multiple inheritance, a class may have any number of immediate superclasses. In both cases, a class may have any number of subclasses.
- Both single and multiple inheritance have advantages and disadvantages.
- Most Smalltalk implementations use single inheritance.
- An abstract class gathers useful shared behavior but is not used to create instances.
- A concrete class is used to create instances.

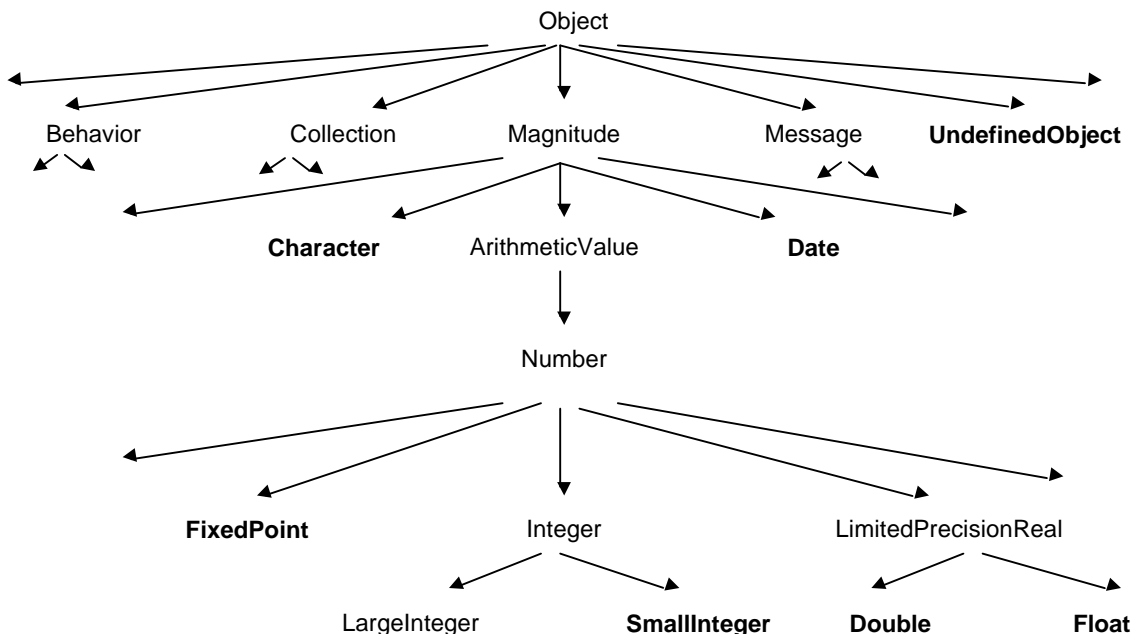
Exercises

1. Consider a program for processing student information. We want to hold the following data on each student: name, address, degree studied, year of registration, courses taken, remaining required courses. Each science student has a computer account number on the 'science computer network', and each arts student has an account on the 'arts network'. Each language student has a password for the language lab, each physical education student has a locker number. All computer science and computer engineering students have a quota on the maximum amount of CPU time on a super computer and their records include the amount of CPU time left. Computer engineering students have a limit on the amount of material they can get for the computer that they are building.
Design a class hierarchy capturing these administrative requirements and list the components of all classes. Explain which classes are concrete and which are abstract.
2. The concept of classification (as in the animal kingdom) is related to the concept of subclassing but not equivalent to it. In particular, one object may be classified as substantially different from another and yet the two objects may be modeled by the same class. As an example, if the only difference between language students and computer science students is that the former use their password to enter language labs and the latter use it to enter computer labs, we may want to model the two kinds of students by the same class. Extend the previous example by adding several new categories of students and data,

- construct a new class hierarchy, and analyze it in terms of the difference between subclassing and classification.
3. Multimedia computers use text, graphics, animated graphics, images, digitized movies, recorded sound, sound calculated from a musical notation, and perhaps smell, taste, and touch in the future. Propose two classifications of all the listed media, one based on single inheritance and one on multiple inheritance.
 4. Given the similarity of animals in the *Farm* program, describe how their implementation could benefit from inheritance.
 5. Which of the following is best characterized as 'is-a-kind-of' relationship and which is a 'has-a' (contains -) relationship? Which of them would be suitable for subclassing (define suitable abstract class if appropriate and draw the class hierarchy) and which of them would best be implemented by *aggregation* - a class with multiple components? Draw a 'containment diagram' and a class hierarchy diagram as appropriate.
 - a. Arm chair, dining chair, desk chair, bed, sofa, love seat, fouton, desk, coffee table, dining table.
 - b. Chair: back rest, back rest frame, back rest cushion, seat, hand rest, screws, seat frame, seat cushion.
 6. Create your own example of subclassing versus aggregation similar to those in the previous exercise.
 7. A subclass normally extends the functionality of its superclass but there are exceptions. As an example from biology, an ostrich is a bird that does not fly and its response to command fly must be redefined as 'sorry, I cannot'. Give two examples in which a subclass must redefine the inherited behavior.
 8. An abstract class often only specifies that all its subclasses should implement a certain behavior but does not implement it itself. To do this, the corresponding method consists of the message subclassResponsibility and each subclass must define the method in its own manner. Find three methods that are 'implemented' in this way. (Hint: Use the *references* to command in the *Browse* command of the *Visual Launcher* and specify subclassResponsibility as the name of the method.)

1.10 Smalltalk's class hierarchy

We already noted that VisualWorks Smalltalk contains a very large library of built-in classes and that the user can access their definitions, modify them, delete them, and add new classes. The classes in the library use single inheritance. At the top of the hierarchy (Figure 1.30) is a single class called Object and all other classes inherit all of its properties. Our diagram shows a minute part of the hierarchy with some of the classes implementing numbers. The diagram shows that class *SmallInteger*, for example, is at the sixth level of depth which indicates that it inherits much functionality from its superclasses.



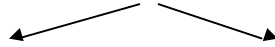


Figure 1.30. A very small part of VisualWorks class hierarchy tree. Concrete classes are shown in **boldface**, all other classes are abstract.

To find out about Smalltalk's class hierarchy, use the *System Browser*. After selecting a class such as *Magnitude*, select the *hierarchy* command in the class view <operate> menu and the text view will display as in Figure 1.31, showing subclasses and superclasses and all instance variables.

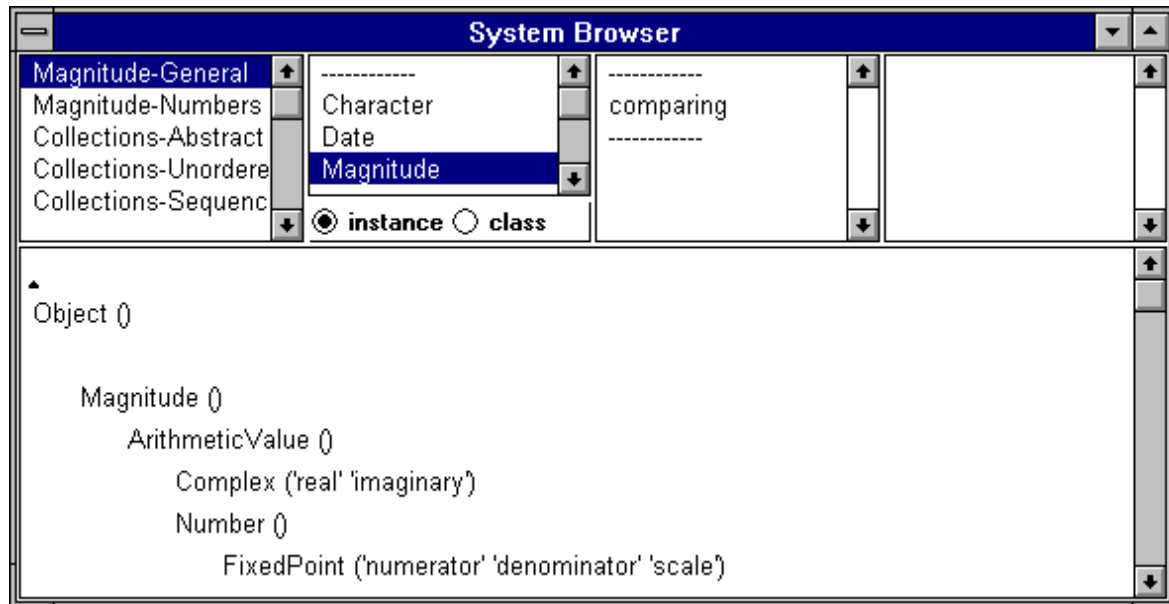


Figure 1.31. System Browser showing a part of the hierarchy of class *Magnitude*.

The hierarchy shown in Figure 1.31 shows both abstract and concrete classes. Class *Magnitude* is abstract and factors out properties needed for comparison, and its subclasses include numbers, printable characters, date objects, and time objects. *Magnitude* has no instances because there is no need for such abstract objects. Classes *ArithmeticValue*, *Number*, and *FixedPoint* are also abstract but class *Complex* is concrete and used to create concrete numbers. Classes *Character*, *Date*, and *Time* further down the *Magnitude* hierarchy are also concrete.

In our discussion of subclassing and inheritance, we have so far talked mainly about inheritance of properties without paying attention to the effect of inheritance on the execution of messages. We will now show that inheritance affect *message resolution*, the way in which Smalltalk finds the definition of a message being executed. As we know, when a program sends a message to an object, Smalltalk examines the receiver's class and executes the method if it is defined in this class. If the method is inherited, Smalltalk does not find the definition in the receiver's class, climbs to its superclass and looks for the definition there. If it is found, it is executed, otherwise the climb continues until either a definition is found or executed or until the top of the hierarchy is reached without finding the method. In this last case, Smalltalk maintains control and creates an *exception* which allows the user to assume control and correct the error or terminate the execution of the program. We will deal with this issue in more detail later.

Main lesson learned:

- VisualWorks classes are organized into a single inheritance hierarchy.
- The top of the Smalltalk hierarchy is class Object and all Smalltalk classes inherit all its properties.
- When a message is sent to a receiver, the first definition of the message found going up from the class of the receiver is executed. If no definition of the method is found, Smalltalk creates an exception and passes control to the user.

Exercises

1. Find and print out class hierarchies of the following classes. Use the *instance* side of the browser and list all superclasses and all subclasses but nothing else.
 - a. Character
 - b. Date
 - c. Number
 - d. Text
2. How many instance variables does class Date inherit and how many new instance variables does it define?
3. How many instance and class variables do all classes inherit from Object?

1.11 Polymorphism

The word *polymorphism* means ‘occurring in many shapes or forms’. In object-oriented programming, polymorphism means that different classes may implement a method with the same name and the same basic behavior but the details of the definition are different in each class. As an example, message + is defined in all number classes and in all of them it means the same thing – addition - but its implementation is different in each of them. As another example, every Smalltalk object knows how to construct its textual description in response to message printString but different kinds of objects obviously construct their descriptions differently.

Although the term polymorphism is not used in everyday communication, the concept is not unfamiliar. As an example, when you go to a restaurant and order a soup, you don’t first ask whether the cook is a ‘microwave cook’ or a ‘stove cook’ and place a different order in each case. You simply order a soup and the cook executes the order in the way in which he or she is trained to execute it. Your message thus sounds the same and has the same effect – your soup - but its implementation by different cooks varies. As another example, if you go to a hotel and your room is cold, you change the setting of the thermostat - effectively sending the message ‘change temperature’ - without thinking about whether this activates a gas-based heater, a water-based heater, a wood furnace, or a nuclear fusion-based heater. The same ‘message’ produces ‘the same’ result but how the result is achieved depends on the object that processes the message.

Polymorphism is a very important concept and we will now illustrate it on an example from the programming world.

Use of polymorphic comparison in sorting

Sorting numbers according to magnitude and sorting text according to lexicographic order both depend on the same principle - the ability of the sorted objects to compare themselves with one another. Even a child can sort any collection using the same *algorithm* (step-by-step procedure) if it can take any two of its elements and decide which comes first and which comes second. Even when your sorting needs are more unusual, the principle of sorting remains the same. As an example, if you want to sort a collection of apples according to taste, you can use the same principle as for sorting numbers or text if you have a way to compare the taste of any two apples in the collection.

We conclude that to sort objects, we need only one sorting procedure based on the comparisons of any two objects in the collection. The method uses some form of the ‘precedes’ operation, implemented perhaps as a < message. In a programming language that allows the use of polymorphism to the extent that a

sorting algorithm can work with *any* two objects that understand the < message, we thus need only one sorting method and this single method will be able to sort any collection of objects that can execute (in pairs) a < message - even if it is implemented quite differently in different classes. Smalltalk's polymorphism has this property and this gives it enormous power.

One of the beauties of polymorphism is that the programmer who wrote the sorting method did not have to anticipate all its possible uses - and, in fact, could not - and yet created a tool that solves a very common problem faced by developers of many different applications. Once the method exists, programmers will never have to think about writing a sorting method for their specialized needs - thanks to polymorphism.

Another advantage of polymorphism is that it eliminates the need to test what kind of object we are dealing with when we need to perform a polymorphic operation. As an example, if we want to add together two numbers, we don't have to test which of the many possible varieties the numbers belong to and then execute the appropriate kind of addition. Instead, we simply ask the numbers to do addition and they do it in the way appropriate to their nature. As another example, a drawing program that displays a collection of geometric objects stored in a file can use polymorphism to ask each individual object to display itself, without testing whether the object is a rectangle, a line, an arc, and so on, and then issuing the appropriate message. This simplifies programming, speeds up program execution by eliminating tests, and makes programs very flexible and easily extendible.

Polymorphism and binding

The possibility to have different implementations of the same method in different classes does not, by itself, give a programming language all the power that we described. If a language requires that we name the class of objects that can be used with a particular method, the advantage of polymorphism is severely restricted because each unrelated class then requires its own definition of sorting. Only a language that does not require specification of the class of its objects in the definition of a method can take full advantage of polymorphism. Smalltalk has this ability but some other languages don't.

The disadvantage of the flexible arrangement that we described is that if the definition of a sorting method does not specify which classes of objects it may use, the *compiler* program that converts the definition into computer code cannot decide which definition of < to bind to the code. An instance of any class that has the definition (and all its subclasses) could be the receiver and any of these methods could be executed. The compiler must thus leave the decision as to which < method to use until execution time: If the receiver of the < method is an integer number, then the definition of < for numbers will be used. If the receiver is a string of letters, then the definition of < for strings must be used, if the collection contains dates, we must use the Date definition, and so on. The run-time look up required to make select the appropriate method takes extra time but modern implementations are so efficient that this extra time is negligible in almost all applications.

Languages that leave the decision of which kind of object will execute a message until execution time are said to use *dynamic typing* whereas languages that require that binding be specified at compile time are said to use *static typing*. Smalltalk uses dynamic typing and this is one of the major reasons for its power.

In addition to eliminating method lookup at run time, another advantage of static typing is that the compiler can catch attempts to send messages to objects that do not understand it. Proponents of dynamic bounding will counter that code that contains such basic errors has not been properly designed and inspected.

Main lesson learned:

- The term polymorphism means that a message with the same name and the same purpose is declared in several classes in a way appropriate for the nature of the class.
- Polymorphism often makes it possible to eliminate lengthy tests to decide which message in which class is suitable to perform a given task. With polymorphism, the receiver itself makes the decision by executing its own form of the message.
- When polymorphism is used to eliminate multiple decisions, it speeds up execution.
- Polymorphism makes it possible to create very general solutions that remain valid even when new classes implementing the task are added to the library. In this way, it greatly contributes to reuse.
- Dynamic typing refers to leaving the decision of which object may be the receiver or the argument of a message until run time. Dynamic typing is necessary for full fledged polymorphism.
- Languages with static typing require full specification of the kinds of receiver and arguments in the code. This restricts the use of polymorphism but provides greater security for coding and may result in slightly faster execution.
- Smalltalk uses dynamic typing.
- An algorithm is a precise description of a sequence of steps that must be executed to solve a problem.

Exercises

1. Find and count all definitions ('implementations') of the following methods.
 - a. `displayOn:`
 - b. `squared`
 - c. `<`
 - d. `=`
2. Give an example of polymorphism from the physical world.
3. The three animals in the *Farm* world share the following messages: `color`, `name`, `eat`, `isHungry`, `run`, `walk`. Some of them are implemented in the same way in each animal, others are not and the program executes them polymorphically. List each group on the basis of your intuition.

Conclusion

The principle of object-oriented problem solving is solving problems by building computer models of miniature worlds populated by interacting objects. Although this paradigm has been applied in a variety of settings, its major use is in computer programming with object-oriented programming languages. Object-oriented programming languages support the concept of a library of objects and provide an environment with tools for creating, deleting, and modifying them.

For the purpose of programming, an object can be thought of as a command-obeying robot specializing in a well-defined behavior and maintaining specialized knowledge. Another possible view is to think of each object as a specialized computer. An object-oriented program can then be thought of as a multitude of cooperating robots or specialized computers.

A useful introduction to the principle of object orientation is to construct scenarios and conversations - sequences of events that typically occur in the problem world at hand. Scenarios are also used by program developers to discover objects required to solve a given problem. Once these objects are found and their properties described, their representation in the selected programming language gives the computer solution of the problem.

The two basic properties of objects are that they understand messages (requests to perform a service) and that they have an internal state. In Smalltalk, all messages return an object - even if the sender of the message does not need it.

Objects can be divided into two groups. Objects that manufacture other objects are called classes, objects manufactured by classes are called their instances. A class and its instances are different objects and understand different messages. Messages understood by classes are called class messages, messages

understood by instances are called instance messages. A detailed definition of the computation performed by a message is called a method.

Since all work in a pure object-oriented language is achieved by sending messages, method definitions themselves consist of message sends. Some of the messages may be directed at the object executing the method (the receiver itself) and others may be directed at other objects. Foreign objects that help in the execution of a method are called collaborators and sending messages to collaborators and taking advantage of their functionality is called delegation.

In addition to functionality, objects have a state but the state is hidden inside the object and can only be accessed by messages. An object is thus an encapsulation of state and functionality. In this, objects resemble electronic chips whose functionality and state are encapsulated inside the chip in some unknown, invisible, and ultimately irrelevant way, and whose only connection to the outside world is via signals transmitted via chip pins.

The state of an instance of a class is stored in its instance variables, the state of the class object is stored in its class variables. The complete description of the functionality of an object (its methods) and a list of its state variables are stored in its class definition. The definition may also include class variables and class methods.

In addition to instance and class variables, OO languages use other kinds of variables that will be introduced later. The common properties of all variables are that at any time, a variable refers to a single object but during the execution of a program the state of this object or even its identity may change. Note also that although one variable always refers to exactly one object, an object may be referred to by more than one variable at a time.

Classes are organized into hierarchies in which most classes have a superclass and inherit all its properties. Using inheritance, each subclass only needs to define its special features to achieve its distinct behavior. Inheritance is transitive which means that each class inherits all properties of all its superclasses. Subclassing saves work, eliminates duplication - a potential source of mistakes, and enforces shared behavior.

Object-oriented programming languages recognize two forms of inheritance - single inheritance and multiple inheritance. In single inheritance, a class may only have one direct superclass; in multiple inheritance, a class may have several superclasses. Graphically, single inheritance results in a class hierarchy that looks like an upside-down tree whereas multiple inheritance may look like a web with crossing links.

The advantage of single inheritance is its relative simplicity. Its disadvantage is that objects sometimes require properties of classes that appear on disjoint branches of the hierarchy tree and some of the properties must then be redefined and uplicated. Multiple inheritance does not suffer from this problem but its disadvantage is that if the inheritance pattern is not trivial, it may be difficult to understand its implications, especially when different superclasses contain similar properties and behaviors. Multiple inheritance also makes it more difficult to control the implications of changes in higher level classes. Most Smalltalk implementations use single inheritance, taking the view that simplicity is more important than occasional duplication. The class at the top of Smalltalk hierarchy is called Object and all Smalltalk classes inherit all its behavior.

Inheritance causes subclasses to inherit both properties and behaviors. It also affects message resolution - the process of finding the appropriate definition of a message. Message resolution consists of looking at the class of the receiver and looking for the method defining the current message, searching superclasses in upward order if the method is not found. A failure to find the method causes an exception

Some superclasses only factor out shared properties and behaviors of their subclasses and are not used as 'object factories'. Such classes are called abstract whereas classes designed for instantiation are called concrete. Although a superclass is usually abstract, it need not be - concrete superclasses exist too. Similarly, a class at the end of a branch need not be concrete although such an arrangement would not make sense unless the class is defined as a starting point for future speciliazation.

Smalltalk classes are organized into categories - groups related by their purpose - and methods in a class are similarly grouped into protocols. The concepts of categories and protocols are purely organizational and have no effect on class behavior and no relation to class hierarchy.

One of the essential concepts of object-oriented programming is polymorphism. Polymorphism means that several different classes define a method with the same name and the same purpose, and each implements it in a way appropriate for its particular character. Polymorphism makes it possible to create

methods with very general applicability, and designs that derive their simplicity from the delegation of decisions to objects performing specialized tasks. Polymorphism facilitates distribution of intelligence among many classes, a design style generally preferred to designs with fewer highly intelligent classes. It also greatly simplifies programs and increases the potential for reuse.

The power of polymorphism is related to the implementation of typing - the time at which it is decided which definition of a method will be used to execute a message. In Smalltalk, this decision is made when the message is actually sent, at run time because the kind of object sending the message is not specified in the program. This kind of typing is called dynamic typing and its advantage is that it allows programs to take full advantage of polymorphism. One disadvantage of dynamic typing is that it prevents compile-time checks whether the receiver of a message and the message match, but this should not be a problem in a properly designed and tested program. Another disadvantage of dynamic typing is that the run time decision as to which method definition will be executed requires extra work on the part of the computer. Current execution techniques make this overhead negligible.

Some programming languages require or encourage their users to write programs so that the compiler can determine typing before the program is executed. This typing style, called static typing, eliminates the overhead of dynamic typing and allows prevention of errors due to mismatched receivers and messages. Its disadvantage is that it severely curtails the power of polymorphism.

Terms introduced in this chapter

abstract class - class declared for the purpose of factoring out properties and behavior shared by a group of related subclasses; not intended for instantiation (see also *concrete class*)

algorithm - orderly sequence of steps describing unambiguously how to solve a problem

argument - a value supplied with a message to make possible its execution

Browser - Smalltalk tool that allows viewing, creation, and editing of Smalltalk classes

class - an object factory, an object that manufactures instances according to a blueprint stored in its definition (*concrete class*); alternatively, a class factoring out shared properties of concrete classes (*abstract class*)

class message - message understood by a class (see also *instance message*)

class variable - holder of a state property of a class (see also *instance variable*)

compiler - a computer program that converts code written in a programming language to directly executable CPU level

conversation - a sequence of events that occur during the execution of a *scenario*

CRC card - Class-Responsibilities-Collaborator card; a card with the name of a class, its description, a list of its responsibilities, and a list of collaborators needed to fulfill them

class definition - formal statement of class properties in the selected programming language

class hierarchy - arrangement of classes in a subclass - superclass relationship

collaborator - an object whose functionality is needed to implement a class responsibility

concrete class - object factory, class designed to be instantiated (see also *abstract class*)

delegation - forwarding execution of tasks that are outside the realm of responsibilities of the receiver to objects equipped to deal with them

dynamic typing - leaving the decision as to which method definition will be used to execute a message until run time

encapsulation - gathering of state and functionality into an object

exception - an illegal event such as attempt to divide by zero or to execute a message not understood by the receiver

functionality - collection of services available from an object; the set of messages that it understands

information hiding - making state information accessible only through explicitly defined messages

inheritance - access to state variables and functionality declared in a superclass

instance - object created by sending a creation message to a class

instance message - message understood by an instance of a class (see also *class message*)

instance variable - holder of a state property of a class instance (see also *class variable*)

instantiation - the act of creating an instance

message - request for service sent by one object to another; each Smalltalk message returns an object

message interface - the collection of all messages that an object understands

message resolution - the process of finding the definition of a message prior to its execution

method - definition of a message; message is what you send, method is a formal description of how it works

multiple inheritance - inheritance where each class may have more than one direct superclass (see also *single inheritance*)

object - an entity with state and functionality

<operate> button - the middle button of a three-button mouse - displays numerous commands including text editing commands such as *copy*, and code execution commands such as *do it*

<operate> menu - pop up menu of the *<operate> button*

polymorphism - ability to define a method with the same name and the same purpose in more than one class

protocol - a grouping of related methods in a class definition

receiver - object to which a message is sent

responsibility - ability to perform a service

scenario - task typically executed in a given problem setting

<select> button - the leftmost button of the mouse; used to select text and other screen items

single inheritance - inheritance where each class may have only one direct superclass; the norm in Smalltalk (see also *multiple inheritance*)

static typing - deciding which method definition will be used to execute a message during compilation (see also *dynamic typing*)

subclass - class at a lower level but on the same branch of the hierarchy tree

superclass - class at a higher level but on the same branch of the hierarchy tree

variable – a named reference to an object

widget - a window component such as a button, a label, or a scrollable selection list

Chapter 2 - Finding objects

Overview

In Chapter 1, we introduced the principles of object-oriented problem solving but we have not paid much attention to how object-oriented solutions work. And we completely ignored the question of how to solve a problem using this approach, in other words, how to find the objects. Although there is no magic formula, a lot of experience gathered by experts has been crystallized into procedures that help in this difficult task and we will describe a simplified form of one such procedure and illustrate it on examples.

This chapter is an introduction to design so don't expect to learn it here and don't be frustrated if you don't. Design must be practiced and we will provide much more exposure as soon as we know some Smalltalk.

2.1 Examples of object-based solutions

Now that we know the nature and the role of objects, it is time to look at whole object-based solutions and how they can be constructed. Before we introduce a method for constructing object-oriented solutions, we will now present two problems and show their solutions, without showing how they have been obtained. We will limit ourselves to the general features, showing the main classes, their relationship, and their cooperation during the operation of the application. The presented solutions are not complete because our goal is to demonstrate their general nature rather than all the underlying detail.

Example 1: Front-end of a chess playing program

Chess playing programs are among the most difficult problems and we will not attempt to attack this complex task in this section. Instead, we will focus on the 'front-end' of such a program, the part that displays the chessboard and knows how to recognize and display legal moves. This part of the chess program does not possess any intelligence beyond knowledge of chess rules and the current state of the chess board. The desired user interface is shown in Figure 2.1 and the complete implementation is presented in Chapter 13.

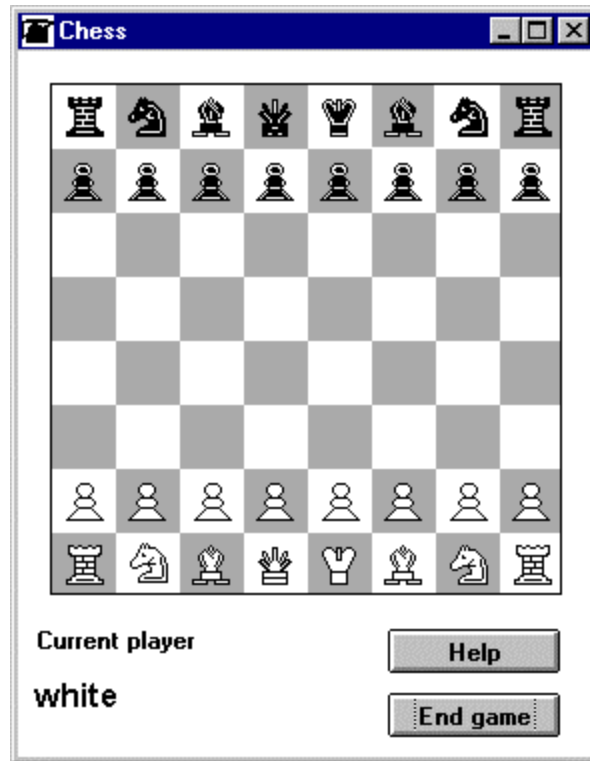


Figure 2.1. Chess user interface.

The program works as follows: When the player clicks a piece and its desired new position, the program first checks whether the move is legal. If it is, it makes the move and displays the new state on the screen. The program displays a message when the player attempts to make a move that is illegal or that results in a check to the opponent's king, a check mate, or a draw.

Solution: A solution consists of classes that collaborate to accomplish the specified goal. The essence of our solution is a class representing the chess board (`ChessBoard`) and one class for each kind of piece (King, Queen, and so on). All chess pieces share certain basic properties and these properties are collected in an abstract superclass called `ChessPiece` (Figure 2.1).

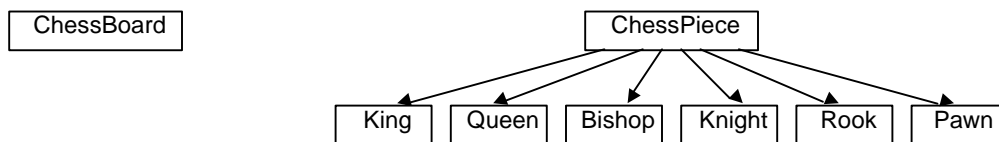


Figure 2.1. Class hierarchy of chess classes in Example 1.

A brief description of some of these classes is as follows:

Class `ChessBoard` keeps information about each square of the board and knows whether it is occupied and by which piece. It also knows how to draw the board at the start of the game, and how to redraw it after a move. In doing so, it assumes that each piece knows how to draw itself.

Class `ChessPiece` knows the color of its piece (black or white). It can also determine whether a move is horizontal, vertical, or diagonal and whether it can be made without colliding with other pieces. These methods are used by its concrete subclasses to test the legality of their particular moves. Since `ChessPiece` is an abstract class, it does not have any instances.

Class King knows how to test whether it can move to a given location (to do this, it uses methods inherited from ChessPiece) and whether the king is in a check position. It knows how to draw itself. Class King is concrete and two instances exist at any time, one for each player.

Class Queen knows how to test whether it can move to a given location and how to draw itself. Class Queen is concrete and the game starts with two instances, one for each player.

Classes implementing other pieces have similar functionality tailored to the rules of their movement.

After this brief description, let's examine a few scenarios to see how our classes work together to implement a chess front end. We will start with the simplest scenarios.

Scenario 1: Clicking an empty square as the starting point of a move

Conversation:

1. *Player* clicks an empty square as the starting point of a move.
2. *aChessBoard* (an instance of class *ChessBoard*) checks the square, finds it empty, and ignores the mouse click.

Scenario 2: Attempt to move owned piece to an illegal destination

Conversation:

1. *Player* clicks a square occupied by her piece as the starting point of a move.
2. *aChessBoard* checks the starting position and finds it occupied by the player's piece. The attempt is legal.
3. *Player* clicks another square as the destination of the move.
4. *aChessBoard* asks piece to attempt the move.
5. piece tests whether it can move to the specified position and finds that it cannot.
6. piece returns a failure result to *aChessBoard*.
7. *aChessBoard* displays a failure message to *player* indicating the cause of the failure.

Scenario 3: Legal attempt to move a piece

Conversation:

1. *Player* clicks a starting position position.
1. *aChessBoard* checks the square and finds it occupied by the player's piece.
2. *Player* clicks another square as the destination of the move.
3. *aChessBoard* asks piece to attempt the move.
4. piece tests whether it can move to the specified position and finds that it can.
5. piece returns success result to *aChessBoard*.
6. *aChessBoard* asks piece to draw itself in the new position.
7. piece draws itself in the indicated position.
8. *aChessBoard* redraws the original position of piece as an empty square.
9. *aChessBoard* asks the opponent's king whether the new position is a check.
10. king tests whether it is in check and returns the result to *aChessBoard*.
11. If the new position is a check, *aChessBoard* displays a message to that effect.

By verifying that all steps of all our conversations use only behaviors listed in our class descriptions, we can eventually confirm that our classes indeed solve the problem.

Example 2: Travel advisory program

Problem: This program will be used by travel agents to inform customers about available trips. When started, the program displays a window showing all available destinations. When the client selects a destination, another window opens providing access to details about the selection. It contains a 'notebook' widget with tags providing access to information about flights to this location, the location itself, and local car rental companies and hotels. By clicking a tag, the user opens a page showing more information about the selected topic

When the user clicks the location tag, the page displays textual information and a picture of the location. The car rental page is selected in the same way and shows a list of car rental companies available at that location; when the user selects one, the page displays a list of all available rates. For flights, the page

shows a list of flights to this location with the name of the airline company, flight departure and arrival times, and price. The hotel information page shows a list of hotel names and their ratings (one to five stars). When the user selects a hotel, the page displays available rates (a list of number-of-persons-in-room/price items) and a photograph of the hotel, and shows the hotel's location on the map.

Solution: A possible solution uses the following classes, mostly holding information and providing access to it. There are no inheritance relationships.

Class Application is responsible for the main window with its buttons and other parts of the user interface. Its instance variable `locations` holds `Location` objects. This class has a long list of responsibilities, unlike the following classes which are essentially just information holders.

Class Location has detailed information about a destination. It has instance variables `name` (a text object), `description` (text), `picture` (a picture object), `carRentals` (list of `CarRental` objects), `flights` (list of `Flight` objects), and `hotels` (list of `Hotel` objects).

Class CarRental describes a car rental company. It holds a `name` object with the name of the company and a list of `RentalRate` objects in an instance variable called `rentalRates`.

Class RentalRate describes a specific car rental choice. It holds a `carModel` (text) and a `pricePerDay` numeric value.

Class Flight describes a flight. It holds the `companyName` (text), `departureTime`, `arrivalTime`, and `price`.

Class Hotel describes a hotel. Its instance variables are `rating` (text), `name` (text), `listOfRates` (instances of `HotelRate`), `picture`, and `coordinates` of the hotel on the map (a point object).

Class HotelRate describes a particular hotel rate. Its instance variables are `occupancy` - a number representing the number of occupants, and `rate` - the price of the room at this occupancy rate.

The following selected scenarios shows how this design can handle our problem.

Scenario 1: User starts the application

Conversation:

1. *User* clicks executes a Smalltalk expression or clicks an icon to open the program.
2. `Application` creates its instance (called `anApplication` in the following steps) and reads travel information from a file into variable `locations`.
3. `anApplication` obtains all destination names from the `locations` variable.
4. `anApplication` opens the main window showing all destination names.

Scenario 2: User selects a location and displays information about flights

Conversation:

1. *User* clicks a destination tag.
2. `anApplication` uses `locations` to access the corresponding instance of `Location` (call it `aLocation`).
3. `anApplication` opens a containing information about the selected location and tags for car rentals, flights, hotel reservations, and location information.
4. *User* clicks the *Flights* tab.
5. `anApplication` uses `flights` to obtain information about individual flights. For each `Flight`, it obtains and displays the information on the *Flights* page.

We leave it to you to construct additional scenarios and check that our selection of classes can indeed solve the problem.

Exercises

1. Draw a very simple chess board situation with three or four pieces. Identify the objects involved in this situation and assign each to one student. Each student will make an index card and record all information required by his or her assigned object on it. As an example, the *chess board* person will have a record of the chess board and the occupants of individual squares, and the *king* person will record the king's color. Play scenarios similar to those listed above, making sure to update information on cards as you play.

2. Repeat Exercise 1 but assign each classes instead of pieces. Each card contains the name of the class, its brief description, and a list of its functionalities. Replay the scenarios.
3. Construct detailed conversations involved in several other scenarios in Example 2.
4. Play scenarios from Example 2 in the way described in Exercise 2.

2.2 Finding objects

We have now seen the essentials of object-oriented solutions of two problems and it is time to show how such solutions can be found. We will outline the general principles in this section and illustrate them on two examples in the rest of the chapter.

Methodologies for identifying objects

Although there is no set of rules that would let you find objects for an arbitrary problem mechanically, experienced practitioners formulated sets of steps and guidelines that help in finding a solution. These recipes are called methodologies.

Since the emergence of object-oriented programming, several methodologies summarizing such steps and guidelines have been formulated. In this book, we will use a methodology loosely based on Responsibility-Driven Design (RDD) originally described by Rebecca Wirfs-Brock and her co-workers. In this section, we will explain the principles of RDD and the following sections will illustrate the method on two examples. For further details, read the specialized literature listed in the References at the end of the book.

Before outlining our approach, we will give a more accurate definition of what a design methodology is. Formally speaking, a design methodology consists of a process, a set of guidelines, and a set of deliverables. A *process* is a sequence of steps that must be executed to solve a task (in this case finding classes for a given problem). The *guidelines* provide hints that should help in the execution of the process and in finding a good solution. *Deliverables* are the artifacts (the documents and the software) that must be produced during the process. Each methodology prescribes the contents of the required documents, usually as a combination of text and diagrams using a set of carefully defined graphical symbols.

A point worth noting is the great importance attached by all methodologies to documentation. Most of these documents take a lot of time to produce and are difficult to keep up to date, and some developers think that producing them is a waste of time and that the only deliverable should be the software. There are even some who claim that orderly design is a waste of time and that one should simply sit down and develop the application using only the programming environment of the selected programming language. (In all justice, the pressure of deadlines sometimes make it difficult to produce both documentation and the product and there is no difficulty in deciding which of them is more important to produce.)

The reason why *orderly design* is desirable is that any non-trivial application quickly becomes unmanageable if you don't develop it systematically. If the number of classes exceeds ten or so, keeping track of them becomes impossible for most ordinary programmers. If you 'hack' the program and create it in a disorderly fashion, you will soon lose track of the purpose of the classes that you created, their mutual relationship, and the purpose and use of their methods and variables. Furthermore, as the project evolves, you will invariably decide to make changes and since you will be reluctant to redesign the application, you will continue making patches to the wrong design. The result will be that you will spend more time on development than if the design was done systematically from the beginning because making design changes before any code has been written is much easier than making changes when your ideas become a program.

The reasons why you need *good documentation* are much the same. Without documentation, the design soon becomes very difficult to understand and further development becomes impossible. Even worse, when the product is completed and delivered, its maintenance will be very difficult if the original design is not documented. Maintenance without documentation is difficult even when the original developers are still available, and almost impossible when they leave the development team. In summary, although proper process and documentation cost extra time and money in the short term, they save time and money in the long run. They also give a better assurance of quality. Like everything else, pre-

implementation analysis and documentation can be overdone and too much time and money spent on it. Critics refer to this mistake as 'analysis paralysis'.

Responsibility Driven Design

All Object-Oriented (OO) methodologies divide application development into the formulation of a detailed initial specification of requirements, refinement of problem understanding and preliminary identification of classes needed to obtain a solution, refinement of class description, and implementation. These stages are usually referred to as requirements specification, analysis, design, and implementation (Figure 2.3). The attention of development methodologies centers around object-oriented analysis and design, frequently referred to as OOA/D.

The general structure of the *development life cycle* of a software product has been inherited from software engineering research preceding the adoption of the object-oriented paradigm. Although this division is still used, the boundaries between analysis and design are fuzzier in OOA/D than they were in classical methodologies. The main reason for this is that analysis and design in traditional languages relied on different paradigms and transition from analysis to design required a transformation from one perspective to another. OOA/D, on the other hand, uses the same paradigm (communicating objects) throughout the process.

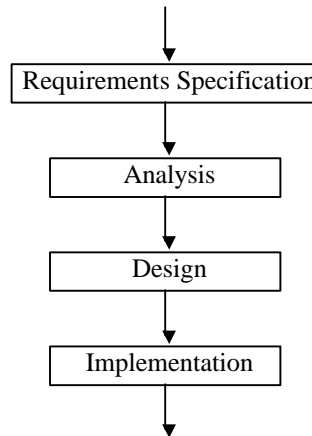


Figure 2.3. Basic software development life cycle.

The development process that we will use and consist of the following major steps:

1. *System Description.* In this step, we obtain a basic problem statement from the client and refine it with the help of subject area expert into a detailed description of the desired product. The specification should avoid, as much as possible, any consideration of the programming language and even the programming paradigm that will be used for implementation.
The deliverables consist of a textual description, a set of usage scenarios and their expansion into high-level conversations between the user and the system, and sketches or computer models of the desired user interface. For more complicated systems or when a larger design is divided among teams, we require a Context Diagram showing the major components of the system that are to be designed and indicating which parts are outside the system, already exist, or will be obtained somewhere else.
If the problem is not trivial, it is usually necessary to create a glossary explaining the terms used in the description.
During this stage, designers and clients closely cooperate to ensure that the specification is complete and all its aspects correctly understood.
2. *Exploratory Design.* Now that we know what the client wants, we study the System Description to find the key classes needed for the solution. We identify candidate classes and write their preliminary descriptions on CRC (Class-Responsibility-Collaborator) cards using the format shown in the following

section. We then expand the high-level conversations from Step 1 into class-level conversations involving the identified classes.

As we develop the conversations, we record the responsibilities that the classes must have to implement their part in the conversations on CRC cards. We also record the attributes that they need to fulfill their responsibilities, and attach names of collaborator classes to individual responsibilities. Finally, we find how and why the classes are related to one another and record this information in an Object Model diagram.

Beginning with this stage, we assume that the solution will be object-oriented. In fact, we even assume a specific programming environment because our choice of new classes depends on classes that are already available. When we are finished, we test the preliminary design informally by executing class-level scenarios with CRC cards and making sure that all conversations can be completed with the recorded information.

The deliverables include class-level scenarios, an Object Model diagram, and preliminary class specifications with class name, brief description, attributes, responsibilities, and collaborators.

3. *Design Refinement.* In this step, we add details to results of Exploratory Design. We check whether any of the previously identified classes are related, and define abstract superclasses factoring out the shared properties if this is the case. We use the results to draw a hierarchy diagram and update the Object Model. We add enough detail to class descriptions to make implementation easy, going as far as deciding the names of methods and describing their roles and principles of implementation. We consider whether any of the identified classes might be useful in future projects and possibly modify the design to make such reuse possible. Finally, we again test whether the identified details are sufficient to satisfy all scenarios.

The deliverables include refined deliverables from Exploratory Design and a class hierarchy diagram.

4. *Implementation.* The detailed design is now translated into a programming language. This step will usually require some additions but if the design was done carefully, these additions will be minimal.

The following table and the diagram in Figure 2.4 summarize the process.

Phase	Activities	Deliverables
System Description	Discuss desired product with client, identify which parts of the system are to be developed and what is outside the system (Context Diagram), find major scenarios and expand them into high-level user-system conversations. Decide preliminary specification of user interfaces. Identify priorities. Write glossary.	Textual specification of requirements, usage scenarios with high-level conversations, model of user interface, Context Diagram. Priority list. Glossary.
Exploratory Design	Analyze specification and scenarios to find candidate classes. Expand high-level conversations into class-level conversations. Use CRC cards for first exploration, convert to electronic form when stable. Test that recorded responsibilities can implement class-level conversations. Identify relationships between classes (Object Model).	CRC cards, preliminary class specifications, class-level conversations, preliminary Object Model diagram.
Design Refinement	Decide whether abstract classes are desirable to factor out shared behaviors. Determine superclass-subclass relationships, decide method names and write descriptions of more complicated methods. Refine Object Model by adding superclass information.	Finalized class specifications, class hierarchy diagram, refined Object Model diagram.
Implementation	Implement design in the selected programming environment. Test functionality using high-level scenarios and other techniques. Evaluate performance and optimize critical parts if necessary. Produce manuals and other user documentation.	Tested application and user documentation.

Although our description is complete, it gives the inaccurate impression that development is a linear sequence in which step n is always completed before step $n+1$, and never repeated. The reality is generally quite different because when we start working on step $n+1$, we often find that the results obtained in step n are incomplete or partially incorrect. Results of step n must then be corrected and this may require corrections or additions to step $n-1$ as well, and so on. While this is quite normal, one should always attempt to complete each step as well as reasonably possible. Do not proceed to the next step lightheartedly, assuming that you will have to return to the previous step anyway.

Another reason why development does not usually proceed in a linear sequence is that it is often better to develop an application *iteratively* (Figure 2.4). In other words, the best approach often is to repeat the whole development sequence several times, progressively attaining higher and higher levels of completeness, perfection, and client satisfaction. The main reason for this is that if the project is not trivial, developers rarely achieve full understanding of their clients' desires after the first meeting. In fact, the clients themselves often cannot initially anticipate all details of their needs and spending much time perfectly implementing an imperfectly understood problem would be a waste.

For larger problems, the best approach is thus to start by focusing on the most important and best understood parts of the specification, and design and implement a prototype without spending more time on the first iteration than necessary. The prototype is demonstrated to the client to obtain corrections and further detail, another iteration is performed, the prototype updated, and so on, until a completely satisfactory implementation is achieved. Experts recommend three or four iterations but the number depends on the complexity of the problem, the experience of the development team, client's understanding of the problem, and other factors.

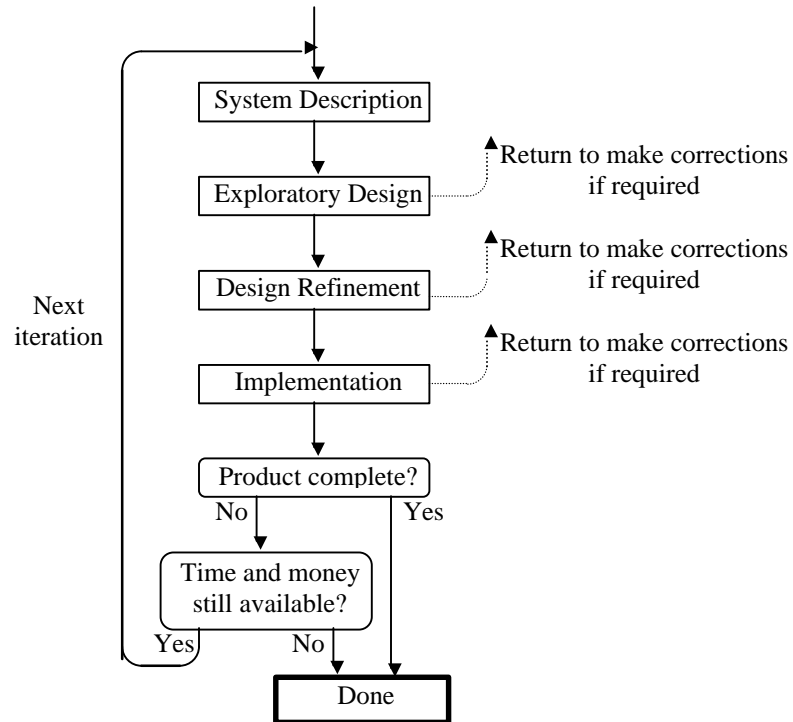


Figure 2.4. Development life cycle. Testing that occurs at each stage is not shown.

Iterative development is the most effective development strategy in terms of time, cost, and client satisfaction and since Smalltalk is excellent for prototyping, it is an ideal choice for this development style. In fact, it has been reported that projects whose target implementation language was other than Smalltalk have been initially prototyped in Smalltalk with the goal of converting the prototype to the target language when a satisfactory level of completion has been achieved. In practice, many such projects were never converted to another language because the developers found that Smalltalk was the best environment to implement the final product as well.

After this general introduction, we will now demonstrate the development process on two examples. Because of lack of space, we cannot fully illustrate the iterative nature of program development and our examples will thus give an inaccurate impression of a relatively smooth linear development process.

Main lessons learned:

- Software development and maintenance consists of a sequence of steps referred as the development life cycle.
- Finding objects and describing the properties that they must have to solve a given problem is the essential part of software development.
- A software development methodology consists of a process, a set of guidelines, and deliverables.
- A process is a sequence of steps to be executed to solve a task.
- Guidelines are experience-based hints that should help to obtain a good result.
- A deliverable is a document or a product that must be completed at a given stage of the development process.
- All software development methodologies use a specification-analysis-design-implementation process.
- In object-oriented development, the boundary between analysis and design is fuzzy.
- Development of a non-trivial application cannot be accomplished in a single pass and it is recommended to use several iterations to achieve better and better understanding of client requirements and optimal design.
- Our methodology is based on responsibility-driven design (RDD). We call its phases system description, preliminary design, design refinement, and implementation.
- Besides textual specification of the problem and a possible model of the user interface, a set of scenarios expanded into conversations is a fundamental part of requirements specification.
- The first decomposition of a scenario is a high-level conversation capturing the dialog between the user and the application. High-level conversations are constructed as a part of system description.
- For complex systems, we construct a context diagram to show major parts of the system and separate those parts that are to be designed from the rest.
- During preliminary design, initial class candidates are derived from the specification. High-level conversations are refined into class-level conversations that show cooperation between classes. A preliminary Object Model showing class relationships is constructed.
- Design refinement consists of finalizing class hierarchy and refining class descriptions and the Object Model.
- All stages of the development process are accompanied by testing.

2.3 Example 1 – A Rental Property Management Program

In this section, we will present our first design example, closely following the development process introduced in the Section 2.2. We suggest that you use our CADE program to create, record, maintain, and print all the required documents. A sample implementation of a more sophisticated version of this problem is available on our web site.

2.3.1 System Description

The program allows a manager to maintain and access information about rental properties and their tenants. Each rental property is a building described by its address and apartments. Each apartment is described by its number, number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of occupants. All information can be updated and saved in a file.

To run the program, the user types and executes a Smalltalk expression and this reads a properties information file and opens the window in Figure 2.5 with a list of all properties arranged alphabetically by their addresses. When the user clicks an address, the apartment list displays the numbers of all apartments in the building. When an apartment is clicked, the window displays all apartment information including the number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of apartment occupants.

Address	Apartment number	Bedrooms	Rent	Tenant Information
Highland 31	111	3	540	Tenant: John MacDonald
Main 254	112			Phone: 542-3782
Main 211	113			Occupants: 3
Prospect 12	114			
Prospect 15	211			
Summer 22	212			

Figure 2.5. Desired user interface.

Given this initial specification, we will now explore the main usage scenarios, presumably with the help of the client. In the first iteration, we will focus on the 'normal' sequences of events and leave the abnormal alternatives (such as what happens when the information file does not exist) for the next iteration.

Scenario 1: *User opens the program.*

High-level conversation:

1. *User* types and executes a Smalltalk expression.
2. *System* reads properties file and displays the window with a list of properties.

Scenario 2: *User closes the program.*

High-level conversation:

1. *User* clicks the window button in the upper right.
2. *System* closes the window and asks whether to save new information.
3. *User* confirms that information should be saved.
4. *System* saves all new information, and terminates the application.

Scenario 3: *User clicks an address in the list.*

High-level conversation:

1. *User* clicks an address in the list.
2. *System* displays list of numbers of apartments in the building.

Scenario 4: *User clicks an apartment number.*

High-level conversation:

1. *User* clicks an apartment number in the list.
2. *System* displays all information available about the apartment.

Scenario 5: *User saves information in file.*

High-level conversation:

1. *User* clicks Save.
2. *System* saves current data about rental properties in a file.

Scenario 6: User changes name of tenant.

High-level conversation:

1. *User* types a new tenant name.
2. *User* clicks *Accept*.
3. *System* updates internal information.

Scenario 7: Users adds a new property.

This has not been described in the specification and the implementation is not obvious. We will invent a suitable implementation and confirm it with the user.

High-level conversation:

1. *User* opens address list <operate> pop up menu showing commands *add* and *remove* and selects *add*.
2. *System* requests property address.
3. *User* enters property address.
4. *System* displays address and requests number of apartments.
5. *User* enters a number.
4. *System* request a number for each apartment, *System* types individual numbers, *User* displays them immediately.

Scenario 8: Users removes an existing property.

This has not been described in the specification either. We will describe a suitable implementation and confirm it with the user.

High-level conversation:

1. *User* selects a property by clicking it.
6. *User* opens address list <operate> pop up menu showing commands *add* and *remove* and selects *remove*.
2. *System* requests confirmation.
3. *User* confirms.
4. *System* deletes the item from the list and blanks all text except for the list of property addresses.

This list of scenarios is obviously incomplete and we leave the rest as an exercise.

Identify major components of the application

Finding the major components of the application will help us understand the system better and clarify which part of the program is our responsibility and what is obtained from other sources.

Reading of the specification suggests several subsystems. One comprises all information about the properties, their apartments, and tenants. This subsystem does not need to know how to display the information or how to accept user input - it is only a computer model of the concepts involved in rental property management. Technically speaking, this subsystem is the *domain model* of our system. Separating the domain model from the user interface is a very good idea because the domain model should be usable with any user interface and any style of access, not only the reading and authoring modes described in our specification. We will call this part of the application the *properties subsystem*.

The next subsystem consists of all classes responsible for displaying a window and its components, and for accepting user input via the mouse and the keyboard. We will call this the *GUI subsystem* because it is responsible for the graphical user interface (GUI). This subsystem does not have any property information but it knows how to access, display, and modify it. It complements the domain model. Since the classes in this subsystem (classes responsible for windows, buttons, other widgets, and user interaction) are independent of the domain that they display, they can be reused in other applications, for example a spreadsheets or a word processors.

We now have a subsystem that contains domain information but does not have a user interface, and a subsystem that can handle all aspects of the user interface but has no domain information. We thus need a subsystem that builds and opens the user interface, provides communication between the user interface and the domain models, and closes the application. As an example of necessary communication between the user interface and the domain model, when the user enters a new apartment, this information must be communicated from the GUI to the domain model. Almost all applications require such a subsystem and the

concept is thus well researched and in the VisualWorks community it is known as the *application model*. We will reuse and extend the built-in facilities and call this subsystem the *property application model*.

A diagram relating the subsystems and the external actors is shown in the Context Diagram in Figure 2.7.

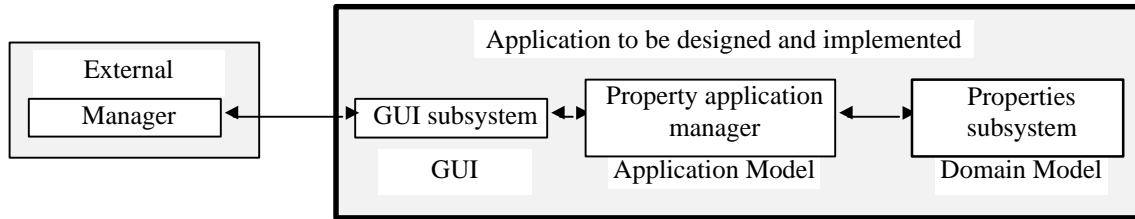


Figure 2.7 Context Diagram showing the main parts of the system.

Glossary

Apartment – information describing an apartment and consisting of its number, monthly rent, number of bedrooms, and tenant.

Manager – person authorized to use the program.

Property – information describing a rental property and consisting of its address and apartment information.

Property application manager – subsystem of program responsible for user interface information and providing a link between the GUI and the domain model.

Properties subsystem – the domain model of the program.

Tenant – information describing a tenant and consisting of name, telephone number, and number of occupants in the apartment..

2.3.2 Preliminary design - find candidate classes and define their responsibilities and collaborators

Identify classes

Identifying classes is the crux of object-oriented development. Unfortunately, as we have already noted, there is no cookbook recipe for finding objects. The following are two of the guidelines formulated by OO experts:

- If you can name it and talk about it, and if it's important to your system, it should probably be an object.
- Don't overload one object with several meanings: One idea – one object.

These two ideas provide conceptual insight by little in the way of identifying objects from the specification. As a more concrete guidance, some experts recommend starting the search for classes by textual analysis of the specification and the scenarios, in other words, by scanning the text for nouns and noun phrases that suggest suitable objects. Nouns that are obviously irrelevant or represent objects that are already a part of the programming environment are ignored. The nouns underlined in the following copy of our specification stand out as potential classes:

The program allows a manager to maintain and access information about rental properties and their tenants. Each rental property is a building described by its address and apartments. Each apartment is described by its number, number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of occupants. All information can be updated and saved in a file.

To run the program, the user types and executes a Smalltalk expression and this reads a properties information file and opens the window in Figure 2.5 with a list of all properties arranged alphabetically by their addresses. When the user clicks an address, the apartment list displays the numbers of all apartments in the building. When an apartment is clicked, the window displays all apartment information including the number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of apartment occupants.

We will now evaluate the desirability of the selected nouns as classes:

- *Address*. If the address included the name of the city, postal code, province or state, and other information, we would implement it as a class. As it is, an address is just a string of alphabetic and numeric characters and strings are already in the library. We thus remove *Address* from our list of candidates.
- *Apartment*. Yes, we will implement this concept as a new class because it represents an object that holds a lot of apartment information together and no related class exists in the library. The class will be called *Apartment* and we will decide shortly what exact meaning it will have.
- *Building*. Another and non-trivial object that does not have a counterpart in the library and will have to be implemented. The class will be called *Building*.
- *File*. The concept of a file is implemented in VisualWorks library and does not require a new class.
- *Name of tenant*. This is just another string and we don't need a class to implement it.
- *List*. This is a special kind of collection and Smalltalk contains several classes that can be used to implement it.
- *Window*. A window is a part of the GUI subsystem and since GUI classes are already in the library, it is beyond our concerns.

We conclude that we need to design and implement the following classes:

- Apartment
- Building

Note that we have eliminated most of our original candidates. That's quite OK because it is better to have more candidates than required rather than miss some important ones. When in doubt, include a candidate and leave the decision whether to keep it or not until further detailed analysis.

Have we missed anything? Yes, we have. We forgot about the tenant and about the application model. The tenant is a rather complex collection of information, and the application model is a non-trivial extension of class *ApplicationModel* in the library. We will call its class *PropertyManager*. The new list is

- Apartment
- Building
- PropertyManager
- Tenant

As you can see, identification of new classes requires a certain knowledge of the anticipated programming environment. Without this knowledge, we could not have concluded that we do not have to design classes to implement lists, buttons, notebooks, files, strings, and other objects. We assumed that we will implement the application in Smalltalk, in fact in VisualWorks Smalltalk. In another version (dialect) of Smalltalk, some of the GUI widgets, for example, might be missing although collection and string classes would be included in any Smalltalk that implements the Smalltalk standard. We also assumed that we have never implemented a similar application and don't have any suitable reusable classes. To this extent, preliminary design and the following stages of the development cycle depend on the selected implementation environment.

Identify class responsibilities

Class responsibilities are determined by the desired functionality, and functionality is described mainly by usage scenarios. We will thus expand high-level scenarios into conversations between classes and record the implied responsibilities as they emerge. When working in a team, we suggest the following procedure:

1. Assign each class to an individual or a small group that will be responsible for maintaining all information about the class.
2. Each group prepares a CRC card (Figure 2.8) for its class. Use 5 x 8" paper index cards.
3. Each group proposes a short description of the purpose of its class, gets it approved by the rest of the team, and writes it on the back of the card.
4. As you work your way through the scenarios, each group records information relevant to its assigned class. If a conversation shows the need for a new class, create a new card and assign it to a new group. (In practice, each individual developer is usually responsible for several classes.)

Class: Building	
Components:	
Responsibilities	Collaborators

I hold all information about a building and provide access to it.

Figure 2.8. The face and the flip side of a CRC card.

We will now go through the first few scenarios, expand high-level conversations into class-level conversations, and update our CRC cards. We leave it to you to check whether our analysis and records are complete, expand the remaining conversations, and fill any gaps found as you test your CRC cards at the end of this stage.

Scenario 1: User opens the program.

High-level conversation:

1. *User* types and executes a Smalltalk expression.
2. *System* reads properties file and displays the window with a list of properties.

Class-level conversation:

1. *User* types and executes the Smalltalk expression `PropertyManager open`.
2. `PropertyManager` reads the properties file.
3. `PropertyManager` obtains addresses from individual properties.
4. `PropertyManager` builds the user interface, and displays the window with the initial information.

As a result of this scenario, we note that `PropertyManager` is responsible for initializing the data and opening the interface. The display of building addresses requires cooperation of `Building`. We record this information on the `PropertyManager` and `Building` CRC cards.

Note that we restrict our attention to new classes and ignore existing classes such as those handling files. This is because our goal is to learn about the classes that we must design.

Scenario 2: User closes the program.

High-level conversation:

1. *User* clicks the window button in the upper right.
2. *System* closes the window and asks whether to save new information.
3. *User* confirms that information should be saved.
4. *System* saves all new information, and terminates the application.

Class-level conversation:

1. *User* clicks the window button in the upper right.
2. `PropertyManager` clicks the window button in the upper right.
5. `PropertyManager` closes the window and asks whether to save new information.

6. *User* confirms that information should be saved.
3. **PropertyManager** saves all new information, and terminates the application.

As a result of this scenario, we note that **PropertyManager** is responsible for closing the interface and saving the data. We record this information on the **PropertyManager** CRC card.

Scenario 3: *User clicks an address in the list.*

High-level conversation:

1. *User* clicks an address in the list.
2. *System* displays list of numbers of apartments in the building.

Class-level conversation:

1. *User* clicks an address in the list.
2. **PropertyManager** detects the selection and uses it to identify the corresponding **Building**.
3. **PropertyManager** requests list of apartments numbers from the selected building.
4. **Building** returns list of numbers.
5. **PropertyManager** displays the numbers in the apartment number list.

As a result of this scenario, we note that **PropertyManager** is responsible for knowing about buildings, tracking selections in the address list, obtaining building information, and updating the apartment number list. We also note that **Building** must provide access to apartment numbers and this is achieved by accessing **Apartment** objects. Finally, each **Apartment** object must be able to access its number. We record all new information on the corresponding CRC cards.

We leave the remaining scenarios as an exercise. As you complete a scenario, don't forget to update the CRC cards. Note that the items starting *User* do not changed much because our focus is expanding the *System* response and identifying how it is implemented by the collaboration of candidate classes.

Assuming that we have completed our analysis of the existing scenarios, we now convert the information recorded on CRC cards into the following format:

Apartment: I hold information about an apartment and provide access to it.

Components:

- number of bedrooms
- rent
- tenant

Responsibilities

- Create new apartment
- Access (modify or return) component information.

Collaborators

Tenant

Building: I hold information about a building and provide access to it.

Components:

- address
- list of apartments

Responsibilities

- Create new apartment
- Access (modify or return) component information.

Collaborators

Apartment

PropertyManager: I am the application model responsible for opening and closing the user interface and propagating information between the user and the domain model. I know about all buildings currently managed

Components: list of buildings

- list of buildings

Responsibilities

- Open application and display existing data.

Collaborators

Building

- Close application and save existing data. Building
- Obtain information about selected building Building
- Obtain information about selected apartment Building

Tenant: I hold information about a tenant and provide access to it..

Components:

- name
- phone number
- number of co-occupants

Responsibilities

- Create new tenant
- Access (modify or return) component information

Collaborators

Draw an Object Model

A picture is worth a thousand words and we conclude with a diagram depicting functional relationships among classes. This Object Model diagram (Figure 2.9) helps us understand why we need the classes that we identified and how they relate to one another. Lines with arrows and legend depict the essence of a relationship between two classes, the * symbol means 'contains zero or more' as in 'a building contains zero or more apartments. When the * is missing, the implication is that exactly one object is being referred to.

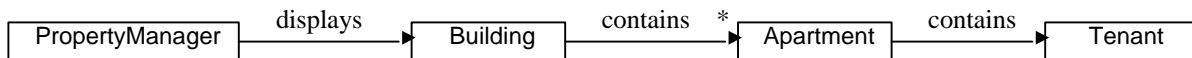


Figure 2.9. Object Model of Electronic Book showing relations between classes.

Testing

To conclude Preliminary Design, we must check that the identified classes and responsibilities are capable of implementing all our scenarios. To do this, redistribute and re-execute the scenarios, making sure that all classes and responsibilities needed to execute all conversations are recorded on the CRC cards. Check that all classes and all responsibilities are used and if not, determine whether they should be deleted or whether scenarios should be corrected or added.

When you are finished testing and the Preliminary Design is confirmed, transfer the information from CRC cards to the format used above and proceed to Design Refinement to identify further details and construct class hierarchies.

2.3.3 Design Refinement

At this stage, we will first check whether we have classes with enough shared behavior to warrant superclasses. We will then draw hierarchy diagrams showing the place of each new class in the class tree, update class specifications making sure that they contain enough detail for implementation, and redraw an Object Diagram if necessary. Finally, we will test that the final model works.

What is the place of our classes in the class hierarchy?

We will now examine all identified classes, check whether any of them share enough behavior to warrant declaring superclasses, and draw a class hierarchy diagram.

All our classes are very closely related but the question is what kind of relationship they have. Texts about OOA/D often distinguish two kinds of relationships: *is-a* and *has-a*. Class A *is-a-kind-of* class B if it is a specialization. As an example, a Lion is a special kind of a Mammal, a Fraction is a special kind of a Number, and a String is a special kind of Collection of objects. Obviously, *is-a* calls for subclassing and this is why a Fraction is a subclass of Number, and why String is a subclass of Collection.

The has-a relationship means containment: A has-a B means that instances of A contain instances of B. As an example, a Car has a Motor and a Chassis, a Fraction has a numerator Integer and a denominator Integer, and a Window has a label, a size, a position, and other properties. A has-a B clearly does not mean subclassing: A Motor is not a special kind of a Car, it's a component of a Car. An integer denominator is not a special kind of Fraction but its part. And a position certainly is not a special kind of a window but one of the necessary components that a Window must have so that it can draw itself.

Considering this perspective, what is the relationship between PropertyManager, Building, Apartment, and Tenant ? We determined that a PropertyManager object knows about the buildings that it manages. The relationship between PropertyManager and Building is thus a has-a relationship – containment and not subclassing. Similarly, an Apartment is not a kind-of building but a building contains an apartment. Apartment is thus a component of Building but not its subclass. And similarly, a Tenant is a part of an Apartment (an Apartment has-a Tenant) but not its special case. Consequently, a Tenant is not a subclass of Apartment but rather its component. As for the rest, the remaining relationships are simply non-existent: A Tenant, for example, is certainly not a special kind of Building, and in our design it is also not a component of a Building. In our design, we decided that a Tenant is a component of Apartment and we will thus not find it listed among the components of Building.

We conclude that there are no is-a (and thus no inheritance) relationship among our classes. But what about some shared meaning or structure? As an example, checking accounts and savings accounts are both accounts (shared meaning) and share a lot of the same properties such as the owner, balance, and number. It thus makes sense that a SavingsAccount is-an Account and so is a CheckingAccount, and both should thus have an class Account for their shared abstract superclass.

In our case, there is no such shared meaning, structure and functionality. The meaning of Apartment is different from the meaning of Building, their components are quite different, and the lists of their responsibilities have very little in common. This means that it does not make sense to define an abstract class to represent the shared meaning and factor out the shared components and behaviors. And the same holds for the remaining classes.

If Building is not a subclass of any new class, then which existing class should be its superclass (because in Smalltalk, every class must have a superclass – except class Object). The classes in the library are mostly system classes such as Number, String, or Window, for use by the environment and their meaning, structure, and responsibilities have nothing in common with Building. Since we have not yet added anything to the class library, we cannot make Building a subclass of any related class of our own and so we decide that Building must be a subclass of Object because Object is the direct or indirect superclass of every class. The same applies to Apartment and Tenant.

The situation with PropertyManager is different because a PropertyManager *is an* application model. This means that PropertyManager must be a subclass of class ApplicationModel.

This completes our investigation of class relationships and our decisions are summarized in the Class Hierarchy Diagram in Figure 2.10. The hashed line indicates additional intermediate classes in VisualWorks library that are not shown. Rectangles with thick border are new classes that we will have to implement, rectangles with thin borders are classes already in VisualWorks library.

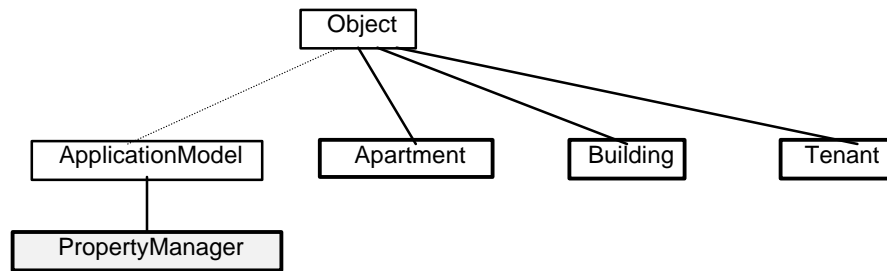


Figure 2.10. Class Hierarchy Diagram for Electronic Book.

Refine class descriptions

Our next task is to examine existing class descriptions and add missing details. The final class descriptions are shown below; we use Smalltalk conventions for naming although they will only be introduced in the next chapter. Note that the names of certain classes such as Integer or Collection, are not quite correct but we don't know better at this point of the book. We are also leaving out certain responsibilities of PropertyManager required for opening and closing the application because they require some knowledge of class ApplicationModel which is covered in Chapter 6.

Apartment: I hold information about an apartment and provide access to it.

Components:

- bedrooms (Integer) number of bedrooms
- rent (Number)
- tenant (Tenant)

Responsibilities

- Create new apartment
 - message new
- Access (modify or return) component information.
 - bedrooms – return the value of bedrooms
 - rent – returns the value of rent
 - tenant – returns the value of tenant
 - bedrooms: anInteger – sets the value of bedrooms to anInteger
 - rent: aNumber – sets the value of rent to anInteger
 - tenant: aTenant – sets the value of tenant to aTenant

Collaborators

Building: I hold information about a building and provide access to it.

Components:

- be address (String)
- apartments (Collection) list of apartments

Responsibilities

- Create new building
 - new
- Access (modify or return) components
 - address – returns the value of address
 - apartments – returns the value of apartments
 - address: aString – sets the value of address to aString
 - apartments: aCollection – sets the value of apartments to aCollection

Collaborators

Apartment

PropertyManager: I am the application model responsible for opening and closing the user interface and propagating information between the user and the domain model. I know about all buildings currently managed

Components:

- buildings (Collection) list of buildings
- We will also need variables supporting the window widgets but this requires some understanding of user interface components in VisualWorks

Responsibilities

- Open application and display existing data.
 - open
- Close application and save existing data.
- Obtain information about selected building
 - buildings – returns the value of buildings
- Obtain information about selected apartment

Collaborators

Building
Building
Building
Building

Tenant: I hold information about a tenant and provide access to it.

Components: name (String), phone (String), occupants (Integer) number of co-occupants

- Close application and save existing data.
- name (String)
- phone (String)
- occupants (Integer) number of co-occupants

Building

Responsibilities

- Access (modify or return) component information
 - name – returns the value of name
 - phone – returns the value of phone
 - occupants – returns the value of occupants
 - name: aString – sets the value of address to aString
 - phone: aString – sets the value of phone to aString
 - occupants: aNumber – sets the value of occupants to aNumber

Collaborators

Refine Object Model

There is no new information because we have not added any new classes and the Object Model diagram thus remains unchanged.

Our design is now complete and we can start implementing it. We are not as optimistic as to assume that we will not have to make any modifications but the design is good enough to allow us to start implementation with a good understanding of what we are doing.

Implementation should now be relatively easy for anybody with sufficient knowledge of VisualWorks Smalltalk. When the prototype is finished, we will show it to the client and obtain corrections and additional details and scenarios. We will then update the design and implementation, and possibly do another specification-analysis-design-implementation iteration. When we are finished, writing user documentation from the detailed specification and scenarios should be easy. It should be noted that some authors recommend writing the manual as early as in the specification phase.

Implementation

An experienced Smalltalk programmer will not have any difficulty implementing our design. Although some details are undoubtedly missing, the overall plan is sound and most of the details resolved.

Conclusion

We started with a rather complete specification, clarified the scenarios and the user interfaces, and identified classes and details of their responsibilities. The procedure was based on progressive refinement, checks, tests, and the use of various perspectives to ensure that no information was missed or left unused. We had to make some decisions that had to be confirmed with the client.

Main lessons learned:

- is-a (or is-a-kind-of) and has-a are two concepts that are very helpful in determining the relationship between two classes.
- A is-a-kind-of B means that class A is a special class of B and should be defined as its subclass.
- A has-a B means that A has an instance of B as its component. No subclassing is implied.

Exercises

1. Execute the example, completing all CRC cards, writing all missing descriptions, and performing all required tests.
2. Read and explain the Object Model diagram.
3. After seeing the prototype, the client has realized that an apartment should have a list of its occupants and that an address should include postal code and city name. Modify the design accordingly. How much impact did the changes have on the original design? Do you think that we should have designed the classes differently?

2.4 Example 2 – The Farm Program

In this section, we will explore how the Farm program introduced in Chapter 1 could be designed.

2.4.1 System Description

The Farm program is a multi-level simulation of a Farm whose purpose is to illustrate certain concepts of object-oriented programming and Smalltalk. Access to individual levels is through a launcher which first opens with Farm 1 selected (Figure 2.12).

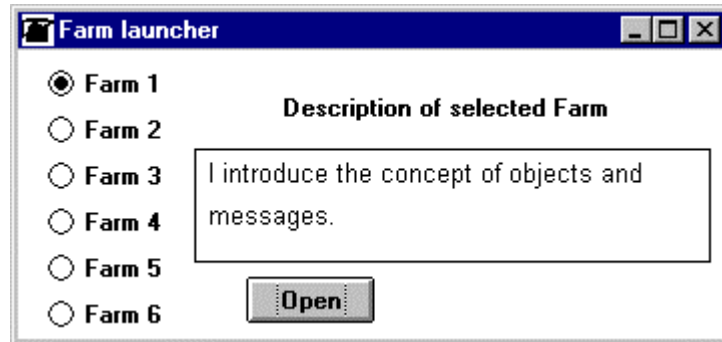


Figure 2.12. Farm Launcher.

When the user selects a farm level and clicks Open, a Farm window as in Figure 2.13 opens.

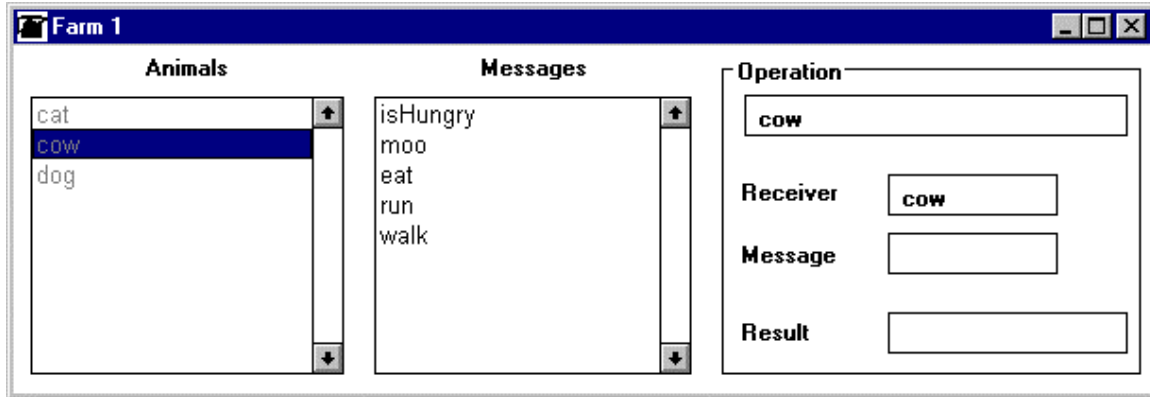


Figure 2.13. Farm 1 window.

The first three levels of Farm share the same user interface, and so do levels 4 and 5. Level 6 uses another user interface.

All six levels are based on three kinds of animals – cat, cow, and dog. In the first level, cat understands commands `isHungry`, `meow`, `eat`, `run`, and `walk`. The cow does not understand `meow` but does understand `moo`, and dog understands `bark` instead. When the user clicks an animal, the window displays the commands that the animal understands, and when the user clicks a command, the window shows the result of executing it. Command `isHungry` produces `true` or `false`, `meow` produces `'I meow'`, `moo` produces `'I moo'`, and `bark` produces `'I bark'`. Command `eat` produces `'hungry = false'`, `run` produces `'I am running'` from cat and dog but `'I am too full'` from cow, and `walk` produces `'I am walking'` from all three animals.

Level 2 interface is the same as for level 1 but the animals understand two new command - commands `color` (returns `'white'` for cat, `'brown'` for cow, and `'black'` for dog) and `name` (`'Pussie'` for cat, `'Jerky'` for cow, and `'Rob'` for dog).

Level 3 introduces three new objects, all of them factories for producing animals: `Cat` is a cat factory, `Cow` is a cow factory, and `Dog` is a dog factory. Each of these understands a single command called `new` and executes it by creating a new animal with name and color specified by user. Consecutive animals of the same kind are numbered consecutively as in `cat1`, `cat2`, and all animals understand the same commands as at level 2.

Levels 4, 5, and 6 perform the same functions as level3 and additional introduce various Smalltalk concepts and generate Smalltalk code. Their design should be considered in general terms now but details should be left for the second iteration.

Scenarios and high-level conversations

Since each level is an extension of the previous level, we don't have to separate scenarios into consecutive levels. We will detail only a few of the numerous scenarios because most scenarios are identical except for the selected command and the corresponding result. The remaining scenarios are left as an exercise.

Scenario 1: User starts Farm.

High-level conversation:

1. User enters and executes a Smalltalk expression.
2. System opens launcher window as in Figure 2.12.

Scenario 2: User closes Farm.

High-level conversation:

1. User clicks the window closing button.
2. System closes all Farm windows and ends the program.

Scenario 3: User opens Farm 1.

High-level conversation:

1. User clicks *Farm 1* radio button.
2. System takes note of the new setting.

3. *User clicks Open.*
4. *System opens Farm 1 user interface.*

Scenario 4: *User sends 'meow' to the cat.*

High-level conversation:

1. *User clicks cat in the Animals list.*
2. *System displays commands understood by cat and updates the rightmost part of the window.*
3. *User clicks meow.*
4. *System displays results in the right-most part of the window, restores the animal list, and erases the list of commands.*

Scenario 5: *User sends 'new' to CatFactory.*

High-level conversation:

1. *User clicks Cat in the Animals list.*
2. *System displays new in the command list and updates the rightmost part of the window.*
3. *User clicks new.*
4. *System requests cat name and color, creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.*

Context Diagram

The information gathered above seems sufficient to identify external actors and subsystems and we use it to construct the Context Diagram in Figure 2.14.

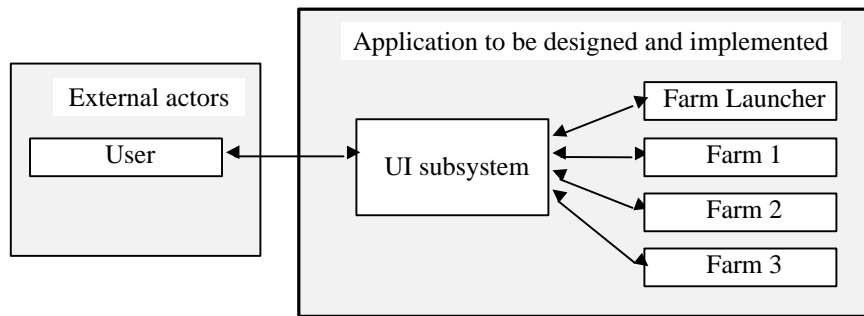


Figure 2.14. Context Diagram of the first iteration of the Farm program.

Glossary

Farm Launcher – main part of Farm providing access to different Farm levels.

Farm 1, Farm 2, Farm 3 – different levels of the Farm program.

2.4.2 Preliminary design

Identify classes

As usual, we start by textual analysis of the specification and the scenarios.

Specification

The Farm program is a multi-level simulation of a Farm whose purpose is to illustrate certain concepts of object-oriented programming and Smalltalk. Access to individual levels is through a launcher which first opens with Farm 1 selected. When the user selects a farm level and clicks Open, a Farm window opens.

The first three levels of Farm share the same user interface, and so do levels 4 and 5. Level 6 uses another user interface.

All six levels are based on three kinds of animals – cat, cow, and dog. In the first level, cat understands commands isHungry, meow, eat, run, and walk. The cow does not understand meow but does understand moo, and dog understands bark instead. When the user clicks an animal, the window displays the commands that the animal understands, and when the user clicks a command, the window shows the result of executing it. Command isHungry produces true or false, meow produces 'I meow', moo produces 'I moo', and bark produces 'I bark'. Command eat produces 'hungry = false', run produces 'I am running' from cat and dog but 'I am too full' from cow, and walk produces 'I am walking' from all three animals.

Level 2 interface is the same as for level 1 but the animals understand two new command - commands color (returns 'white' for cat, 'brown' for cow, and 'black' for dog) and name ('Pussie' for cat, 'Jerky' for cow, and 'Rob' for dog).

Level 3 introduces three new objects, all of them factories for producing animals: Cat is a cat factory, Cow is a cow factory, and Dog is a dog factory. Each of these understands a single command called new and executes it by creating a new animal with name and color specified by user. Consecutive animals of the same kind are numbered consecutively as in cat1, cat2, and all animals understand the same commands as at level 2.

Levels 4, 5, and 6 perform the same functions as level3 and additionally introduce various Smalltalk concepts and generate Smalltalk code. Their design should be considered in general terms now but details should be left for the second iteration.

Next, we examine the selected nouns and add some from our understanding of the situation:

Animal – we don't see any need for this class yet, the program always deals with concrete animals.

Cat – yes, include it; the user interacts with it, it has its own behavior.

Cat Factory – the user interacts with it and it has its own behavior. Yes, include it.

Command – this is a part of each animal's and factory's behavior. Don't include it.

Cow – yes, include for the same reason as Cat.

Cow Factory - include for the same reason as Cat Factory.

Dog - include for the same reason as Cat.

Dog Factory - include for the same reason as Cat Factory.

Farm – according to the specification, we always deal with a specific farm. Don't include.

Farm 1 – include; it has its own user interface and sometimes forces animals to behave differently.

Launcher – include. Has its own user interface and associated behaviors. In fact, it starts the Farm application.

Level 2, etc. – this is just a different name for Farm 2, etc. Include, but under the names Farm 2, etc.

Window – don't include. This is just a reference to the user interface and we have included all user interfaces in the Farm classes and in the Launcher.

We conclude that we will attempt to implement the problem with the following classes:

- Cat
- CatFactory
- Cow
- CowFactory
- Dog
- DogFactory
- Farm1
- Farm2
- Farm3
- FarmLauncher

Identify class responsibilities

We will now examine our scenarios and expand high-level conversations into class-level conversations.

Scenario 1: User starts Farm.

High-level conversation:

1. *User* enters and executes a Smalltalk expression.
2. *System* opens launcher window as in Figure 2.12.

Class-level conversation:

1. *User* enters and executes FarmLauncher open.
2. FarmLauncher opens launcher window as in Figure 2.12.

Scenario 2: User closes Farm.

High-level conversation:

1. *User* clicks the window closing button.
2. *System* closes all Farm windows and ends the program.

Class-level conversation:

1. *User* clicks the window closing button.
2. FarmLauncher closes all Farm windows and ends the program.

Scenario 3: User opens Farm 1.

High-level conversation:

1. *User* clicks Farm 1 radio button.
2. *System* takes note of the new setting.
3. *User* clicks Open.
4. *System* opens Farm 1 user interface.

Class-level conversation:

1. *User* clicks Farm 1 radio button.
2. *System* FarmLauncher takes note of the new setting.
3. *User* clicks Open.
4. FarmLauncher asks Farm1 to open.
5. Farm1 creates an instance of itself and asks each of Cat, Cow, and Dog Factory to create an instance of themselves.
6. Cat, Cow, and Dog to create an instance of themselves.
7. Farm1 asks each potential command receiver for a list of commands that it understands.
8. Cat, Cow, and Dog report their commands. Farm1 asks each potential command receiver for a list of commands that it should not understand. (Farm 1 does not understand color and name.)
9. Cat, Cow, and Dog report commands that they do not understand.
10. Farm1 opens its user interface.

This scenario is perhaps less obvious than you would expect. Our goal in designing it was to deal with possible future modifications. As it is, we can add new animals without changing Farm1 because Farm1 does not depend directly on animals. (Note that adding a new animal still requires a change in the list of receivers that Farm1 contacts but we will not worry about this.) We could also change the set of commands that animals understand or should not understand in Farm 1 and class Farm1 will not have to be changed. Our solution is not, of course, the only one possible.

Scenario 4: User sends 'meow' to the cat.

Comment: Although the resulting behavior is the same for all farms, we must select a specific farm for the purpose of the conversation. Let's select Farm 2.

High-level conversation:

1. *User* clicks cat in the Animals list.
2. *System* displays commands understood by cat and updates the rightmost part of the window.
3. *User* clicks meow.

4. *System* displays results in the right-most part of the window, restores the animal list, and erases the list of commands.

Class-level conversation:

1. *User* clicks *cat* in the *Animals* list.
2. *Farm2* takes note that *cat* is selected as the current animal.
3. *Farm2* displays commands understood by *cat*, and updates the rightmost part of the window.
4. *User* clicks *meow*.
5. *Farm2* requests appropriate response to command *meow* from the current animal (a *Cat*).
6. *Cat* returns the appropriate response and *Farm2* displays it as a part of results in the right-most part of the window, restores the animal list, and erases the list of commands.

Scenario 5: *User sends 'new' to Cat Factory.*

Note: Assume *Farm 3*.

High-level conversation:

1. *User* clicks *Cat* in the *Command Receivers* list.
2. *System* displays *new* in the command list and updates the rightmost part of the window.
3. *User* clicks *new*.
4. *System* requests cat name and color, creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.

Class-level conversation:

1. *User* clicks *Cat* in the *Command Receivers* list.
2. *Farm3* takes note that a *CatFactory* is selected as the current command receiver. It displays the commands that it understands in the command list and updates the rightmost part of the window.
3. *User* clicks *new*.
4. *Farm3* asks *CatFactory* to respond to the message.
5. *CatFactory* requests cat name and color
6. *CatFactory* creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.

With this information, we can now write detailed descriptions of all required classes.

Cat: Simulated cat. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in *Farm1*. A domain object.

Components:

- list of commands understood
- list of commands not understood in *Farm1*

Responsibilities

- Creation
 - Create new an instance of *Cat*.
- Know commands
 - Know list of all commands (color, name, isHungry, run, walk, eat, meow)
 - Know commands forbidden in *Farm1*(color, name)
- Respond to commands
 - Respond to color, name, isHungry, run, walk, eat, meow

Collaborators

CatFactory: Creates new *Cat* instances.

Components:

Responsibilities

- Creation
 - Create new instance of yourself
 - Create new *Cat*.

Collaborators

Cat

Note that the first creation message is used by *Farm3* when it needs a *CatFactory* – it simply creates an instance of itself - a *CatFactory*; this message is a *class* message because we send it to *class*

CatFactory. The second creation message is a request to an existing CarFactory (and *instance* of CarFactory) to create a new cat; since we send it to an instance of CarFactory, it is an *instance* message. It must use Cat as its collaborator because only Cat knows how to create a new instance of itself. The usage diagram in Figure 2.15 further illustrates this distinction: The first bold **new** message is a message to *class* CatFactory (so it is a *class* message) and creates an instance of CatFactory. The second bold **new** message is a message to the previously created *instance* of CatFactory (so it is an *instance* message) and its result is a new instance of Cat.

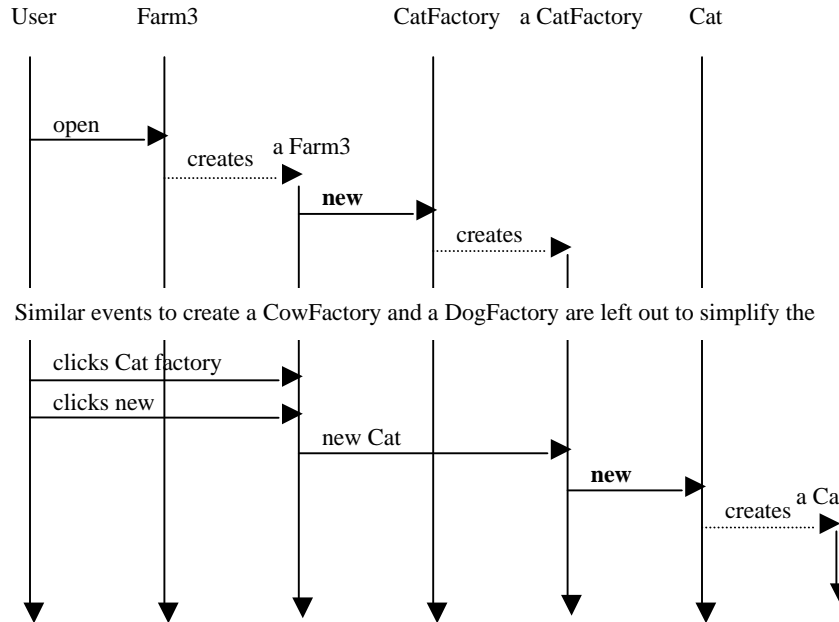


Figure 2.15. Opening *Farm 3* and creating a new cat.

Cow: Simulated cow. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in *Farm 1*. A domain object.

Components:

- list of commands understood
- list of commands not understood in *Farm 1*

Responsibilities

- Creation
 - Create new an instance of Cow.
- Know commands
 - Know list of all commands (color, name, isHungry, run, walk, eat, moo)
 - Know commands forbidden in Farm1(color, name)
- Respond to commands
 - Respond to color, name, isHungry, run, walk, eat, meow

Collaborators

CowFactory: Creates new Cow instances.

Components:

Responsibilities

- Creation
 - Create new instance of yourself
 - Create new Cow

Collaborators

Cow

Dog: Simulated dog. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.

Components:

- list of commands understood
- list of commands forbidden in Farm1

Responsibilities

- Creation
 - Create new an instance of Cat.
- Know commands
 - Know list of all commands (color, name, isHungry, run, walk, eat, bark)
 - Know commands forbidden in Farm1(color, name)
- Respond to commands
 - Respond to color, name, isHungry, run, walk, eat, meow

Collaborators

DogFactory: Creates new Dog instances.

Components:

Responsibilities

- Creation
 - Create new instance of yourself
 - Create new Dog.

Collaborators

Dog

Farm1: Application model, ties user interface to domain objects - input of parameters, output of results.

Components: As required by user interface

Responsibilities

- Opening
 - create and initialize a new instance of yourself, open window
- User interface actions
 - respond to receiver selection
 - respond to command selection

Collaborators

Cat, Cow, Dog

Cat, Cow, Dog

Farm2: Application model, ties user interface to domain objects - input of parameters, output of results.
Same as Farm1 but does not filter out forbidden messages.

Components: As required by user interface.

Responsibilities

- Opening
 - create and initialize a new instance of yourself, open window
- User interface actions
 - respond to receiver selection
 - respond to command selection

Collaborators

Cat, Cow, Dog

Cat, Cow, Dog

Farm3: Application model, ties user interface to domain objects - input of parameters, output of results.
Same as Farm1 but does not filter out forbidden messages.

Components: As required by user interface.

Responsibilities

- Opening
 - create and initialize a new instance of yourself,
open window
- User interface actions
 - respond to receiver selection
 - respond to command selection

Collaborators

CatFactory, CowFactory, DogFactory

CatFactory, CowFactory, DogFactory
Cat, Cow, Dog

FarmLauncher: Application model, provides access to various Farm levels.

Components: As required by user interface.

Responsibilities

Collaborators

- Opening
 - create and initialize a new instance of yourself, open main window
- User interface actions
 - In response to Open, open appropriate farm window

Farm1, Farm2, Farm3

Test our descriptions with all scenarios to check whether they are complete and consistent and make any necessary corrections. Record in a formal format.

Object Diagram

Figure 2.16 shows the relationships between the identified classes.

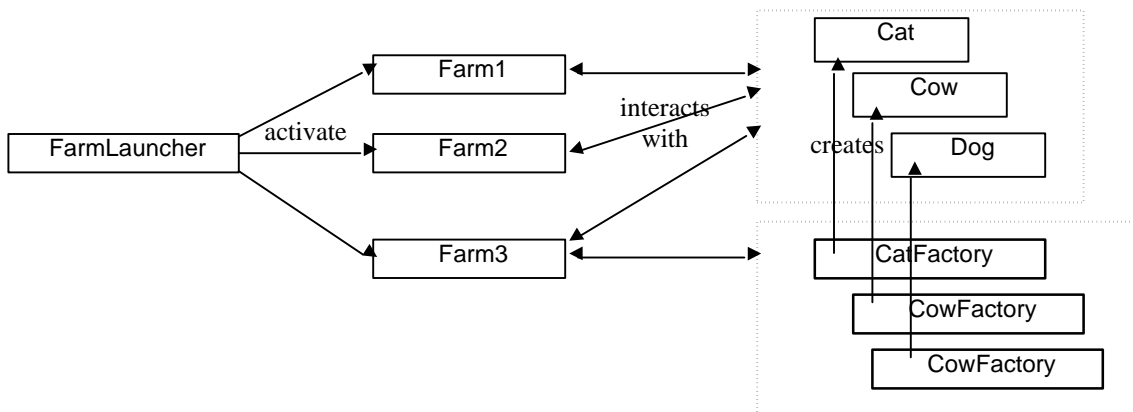


Figure 2.16. Object Diagram of Farm. Only classes to be designed are shown.

2.4.3 Design Refinement

Re-examination of candidate classes and their placeclasses in class hierarchy

Let's examine our candidate classes hierarchically in the order animals, animal factories, farms, and launcher.

The three *animals* are obviously very similar in all important respects: They all are the same kind of objects (animals), they share the same properties (color, name, and isHungry), and they have almost identical functionality - only their form of communication is different (meow, moo, and bark), and they perform some of the same functionality differently (running). The three animals are thus a perfect choice for subclassing and we will make them all subclasses of an abstract class called *Animal*. *Animal* does not have any related classes in the existing hierarchy and it will thus be a subclass of *Object*.

The three *animal factories* are also extremely similar – and simple – their only function is to ask an animal class to create an instance of itself. We don't think that this warrants creating three separate classes – and then a new animal factory class every time we decide to add a new animal. We will thus merge all animal factory classes into one called *AnimalFactory* and generalize its animal creation message so that it can create an animal of any specified class. This solution will make further expansion relatively easy. As an example, if we decide to add a new animal such as sheep, we don't have to create a new factory class to create sheep. *AnimalFactory* does not have any related classes in the existing hierarchy and it will thus be a subclass of *Object*.

When we think about it, we don't actually have to create an instance of *AnimalFactory* when running a Farm program because we never need more than one animal factory at a time. if we use this approach, the animal creation message is always sent to the *AnimalFactory* class and it is thus a *class* message. This is illustrated in the revised diagram of Scenario 3 in Figure 2.17.

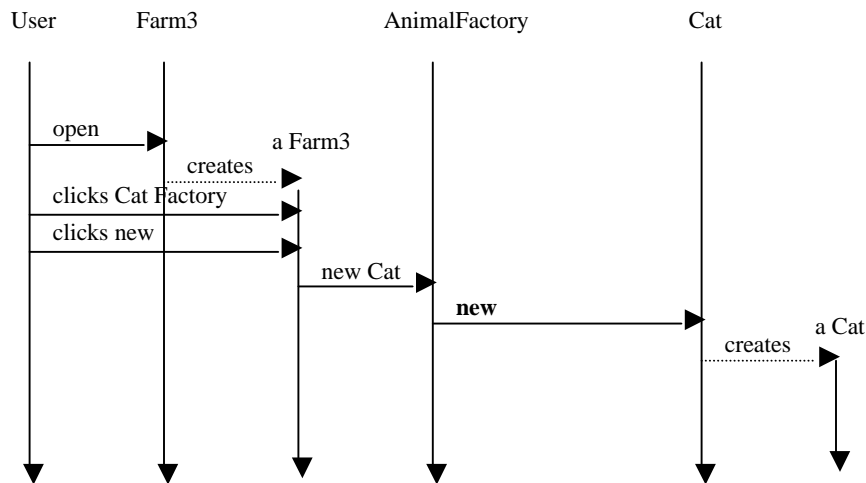


Figure 2.17. Opening *Farm3* and creating a new cat using new design.

We could go even further and decide that since the only function of *AnimalFactory* is to send a message to an animal class to create an instance of itself, this responsibility could be left to the farm. When the user clicks the Cat Factory, for example, the farm could directly send the new message to class *Cat*. This approach is certainly possible but we don't like it because it breaks the uniformity of the behavior of farms: Messages to animals would be sent to animal instances but messages to factories would be executed by the farm. Because of this, we will keep *AnimalFactory* even though its functionality is minimal.

The *farm* category includes three kinds of farms. All are essentially the same – they all have the same interface and the only difference is that *Farm1* understands fewer commands and *Farm3* knows about animal factories and does not open with any animals. This behavior could also be implemented with a single class. However, we must remember that we will need three more kinds of farms and that these will require two different interfaces and three different sets of behaviors. If we wanted to implement all six farms with a single class, this class would be more complicated than we are willing to accept and we will thus implement each farm with its own class.

Now what about the class hierarchy of farms. First, all of them control a user interface so they will all be subclasses of *ApplicationModel*. Class *Farm2* is identical to *Farm1* except that it does not restrict the set of animal commands, and class *Farm3* only extends *Farm2*. We thus decide to make *Farm2* a subclass of *Farm1*, and *Farm3* a subclass of *Farm2*. *Farm1* will define the user interface that *Farm2* and *Farm3* will inherit, as well as the bulk of farm functionality.

Finally the *launcher*. The launcher has a user interface but its purpose, user interface, and behavior is totally unrelated to farms. It will thus stand alone as a subclass of *ApplicationModel*.

The Class Hierarchy Diagram resulting from our analysis is shown in Figure 2.18.

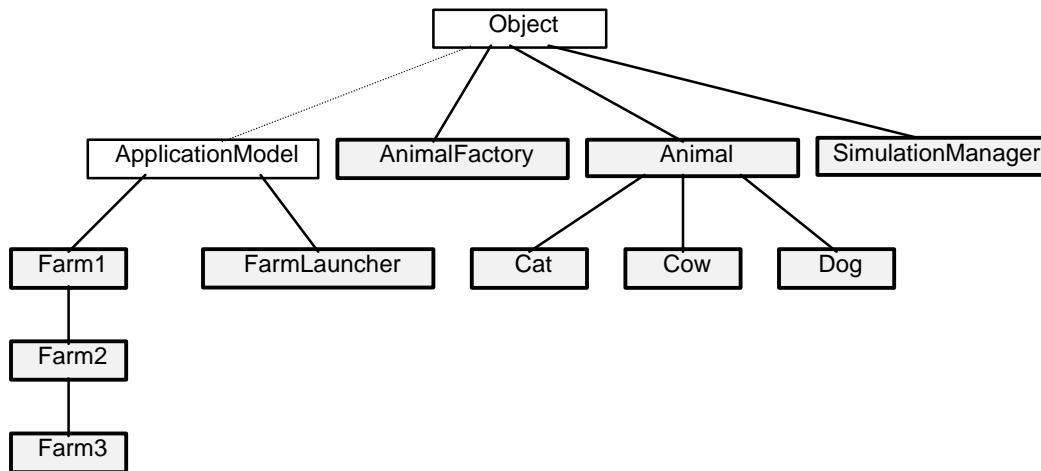


Figure 2.17. Class hierarchy diagram for Farm. Classes that must be constructed have thick borders.

Refinement of class descriptions

After this analysis, we are now ready for the final class descriptions:

Animal: Abstract superclass of all animals.

Superclass: Object

- *Components:* color (string), name (string), isHungry (true or false), commands (list of commands understood), privateCommands (list of commands not understood in Farm1)

Responsibilities

Collaborators

- Creation
 - Create new instance of yourself
- Respond to shared commands
 - Methods color, name, isHungry, walk, eat
- Know names of all shared commands
 - commands – returns (color, name, run, walk, eat)
- Know commands forbidden in Farm1(color, name, isHungry)

AnimalFactory: Creates new animals.

Superclass: Object

Components:

Responsibilities

Collaborators

- Creation
 - Create new animal
 - method newAnimal: anAnimal – class method

Cat, Cow, Dog

Cat: Simulated cat. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.

Superclass: Object

Components: inherited

Responsibilities

Collaborators

- Creation - inherited
- Know commands
 - Know list of all commands
 - commands – add meow to inherited list
- Respond to private commands and commands behaving differently
 - Methods meow, run

Cow: Simulated cow. Knows the commands it understands and how to respond to them. Knows commands that it should not understand in Farm1. A domain object.

Superclass: Object

Components: inherited

Responsibilities

Collaborators

- Creation - inherited
- Know commands
 - Know list of all commands
 - commands – add moo to inherited list
- Respond to private commands and commands behaving differently
 - Methods moo, run

Dog: Simulated dog. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.

Superclass: Object

Components: inherited

Responsibilities

Collaborators

- Creation - inherited
- Know commands
 - Know list of all commands
 - commands – add bark to inherited list
- Respond to private commands and commands behaving differently
 - Methods bark, run

Farm1: Application model, ties user interface to domain objects - input of parameters, output of results.

Superclass: ApplicationModel

Components: As required by user interface

Responsibilities

Collaborators

- Opening
 - create and initialize a new instance of yourself, open window
- Define user interface
- User interface actions
 - respond to receiver selection
 - respond to command selection

Cat, Cow, Dog

Animal, Cat, Cow, Dog

Farm2: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.

Superclass: Farm1

Components: As required by user interface.

Responsibilities

Collaborators

- Opening
 - expand inherited initialization
- User interface actions
 - respond to receiver selection inherited - inherited
 - respond to command selection - inherited

Farm3: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.

Superclass: Farm2

Components: As required by user interface.

Responsibilities

Collaborators

- Opening

- create and initialize a new instance of yourself, open window
- User interface actions
 - respond to receiver selection - inherited
 - respond to command selection - inherited but depends also on

CatFactory, CowFactory, DogFactory

AnimalFactory

FarmLauncher: Application model, provides access to various Farm levels.

Superclass: ApplicationModel

Components: As required by user interface.

Responsibilities

Collaborators

- Opening
 - create and initialize a new instance of yourself, open main window
- User interface actions
 - In response to Open, open appropriate farm window

Farm1, Farm2, Farm3

Object Model Diagram

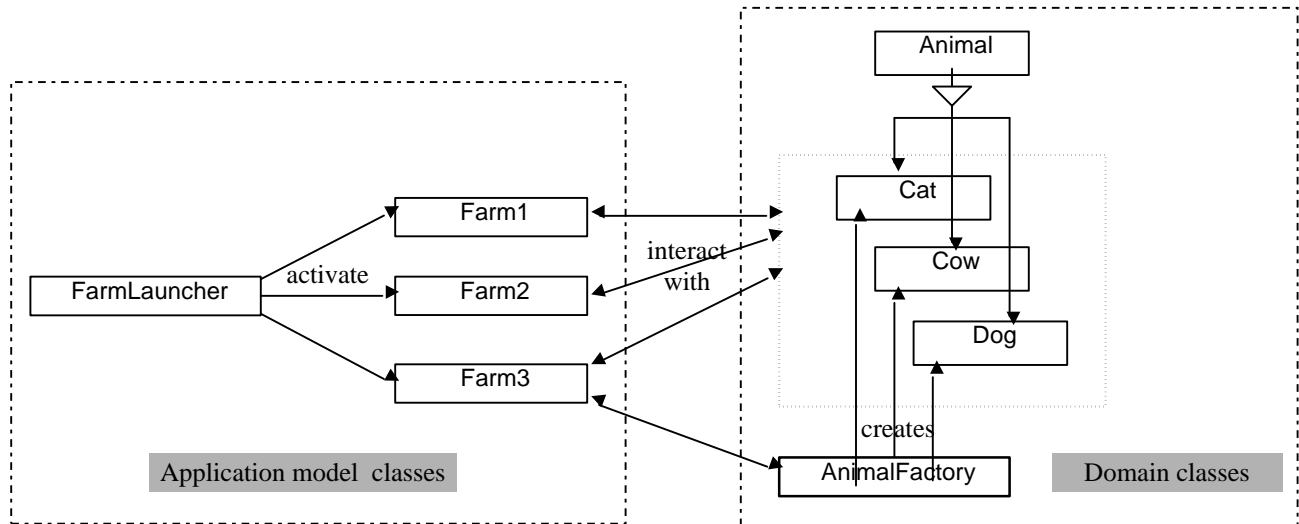


Figure 2.19. Final Object Model diagram. The rectangle below class Animal means that Animal is a superclass of Cat, Cow, and Dog.

Implementation

Our plan for the solution of the problem is now clear and detailed enough and an experienced Smalltalk programmer will not have any problem implementing it. There are, undoubtedly, some details that we have not considered and that will have to be added, possibly even some mistakes that will emerge during implementation, but these will not be difficult to add and they will not affect the overall plan.

In closing

We must admit that our own original design was different because we lumped Cat and CatFactory into Cat, Cow and CowFactory into Cow, and Dog and DogFactory into Dog. This worked but animals and animal factory are more naturally thought of as different objects. Since our experience tells us that it is almost always better to design according to the natural logic of objects, we consider our final design better.

Main lessons learned:

- The initial specification is often vague and incomplete and may require a substantial extension.
- Don't follow the idea of extracting nouns from the specification slavishly. Use any insights that are available such as diagrams and sketches of user interfaces. Use your past experience.
- Watch for possibilities to generalize your design for reuse.
- It is usually possible to implement the same problem in several different ways. Try to use the design that separates the objects in the most natural way.
- Superclasses are usually abstract but there are exceptions.
- Most of the time, we create instances to instantiate behaviors. There are, however, situations in which all the desired behavior can be implemented by the class itself.

Exercises

1. Repeat the example, completing all CRC cards, writing all missing scenarios, descriptions, and performing all required tests.
2. Read and explain the Object Model diagram.
3. Write your own detailed specification and perform preliminary and final design of the following problems:
 - a. Video store inventory program. Keeps track of customers and videos.
 - b. Checkers game.

Conclusion

The essence of object-oriented software development is finding objects and identifying their responsibilities. Although there does not exist a set of rules that guarantees finding the required objects mechanically, a number of methodologies are available that make the task easier and improve the chances of obtaining a satisfactory result.

A methodology consists of a process, guidelines, and a description of deliverables. A process is a sequence of steps that should be followed to solve the problem. Guidelines capture experiences that may help in finding solutions. A deliverable is an artifact, a document or a software product that must be delivered at the end of a process step. Each methodology lists its deliverables and describes their format.

All development methodologies agree that the basic stages of the development process are requirements specification, analysis, design, and implementation. The acronym OOA/D is used to refer to object-oriented analysis and design, the two essential and most difficult stages of object-oriented development. The boundary between analysis and design in object-oriented development is often fuzzy because they both use the same perspective (objects and messages) and the same notations.

The description of software development as specification-analysis-design-implementation hides the fact that all development must be accompanied by constant testing, and that development constantly uncovers gaps and errors that require corrections of results and deliverables produced by previous stages.

In most practical situations, specification, analysis, design, and implementation are not performed only once. Instead, the whole procedure is repeated several times to obtain better and better understanding of the desired product and to come up with better and better designs. Object-oriented design thus has iterative nature.

The methodology that we use is rather loose and we leave it to you to adjust it to your personal needs. It can be summarized as follows:

Requirements Specification:

- Obtain textual description and possibly a model of the user interface.
- Obtain usage scenarios describing major tasks typically performed by the user.
- Decompose scenarios into high-level conversations describing how a scenario is accomplished as a dialog between the user and the system.

- Construct a Context Diagram showing external actors (outside the scope of current project) and major parts of the system to be developed.
- Construct a Glossary of terms. This Glossary will be continuously updated during the process.

Preliminary Design:

- Use results of Requirements Specification to find candidate classes. Textual analysis of the specification, scenarios, and conversations is the basis but not the only resource for this process.
- Expand high-level scenarios into class-level scenarios describing how system actions are performed in terms of dialogs between classes candidates.
- Use class-level scenarios to develop preliminary class descriptions consisting of a brief outline of the purpose of each class, a preliminary list of its components, and a list of its responsibilities. For each responsibility, identify classes that must collaborate to fulfill the responsibility. Record preliminary versions on CRC cards.
- Draw an Object Model diagram showing the essential relationships between the identified classes.

Design Refinement

- Determine whether any of the identified classes share responsibilities that can be factored out. If so, create abstract classes containing these shared responsibilities. Use is-a-kind-of and has-a relationships to distinguish between situations that require subclassing and situations that require containment.
- Draw a Class Hierarchy Diagram showing the place of all identified classes in the existing class hierarchy.
- Refine class descriptions, refine responsibilities into methods, choose method names, and write a short description of each non-trivial method.
- Revise Object Model diagram and refine it to show subclassing.

Implementation

- Implement detailed class descriptions obtained in the previous stage in the selected programming environment. Write user documentation.

Each step is accompanied by testing.

A methodology is not a 'silver bullet' that kills the problem of design and makes design effortless. It is just a set of rules and hints that should help you arrive at a satisfactory solution and produce documentation useful for further development and maintenance.

Finally, it is important to remember that there is no single correct solution and that different designers may produce different working designs. This does not mean, however, that all designs are equally good. The best measures of the quality of design are its extendibility, its maintainability, its understandability, and its potential for reuse in other applications.

Terms introduced in this chapter

analysis - development stage following specification and preceding design

artifact – a product such as a document or a program

class-level conversation - conversation expressed as an exchange between the user and the classes implementing the application

Context Diagram - diagram showing which parts of the application are outside the system but collaborate with it, and which parts of the application are to be designed

design - third stage of development resulting in detailed description of classes; followed by implementation

deliverable - a product that must be delivered at a certain point of the development process

has-a - relationship that identifies a class as a component of another class

high-level conversation - conversation expressed in terms of a dialog between the user and the system without consideration of classes

is-a – relationship that identifies a class as a special kind of another class; also *is-a-kind-of*

iterative development - repeated specification-analysis-design-implementation performed to obtain complete understanding of client needs and optimal implementation

life cycle – the sequence of development, usage, and maintenance of a product

methodology - a process, a set of guidelines, and a description of deliverables

Object Model Diagram - a diagram showing the major functional relationships between classes

OOA/D - object-oriented analysis and design

process - sequence of steps to be executed to perform a task

Responsibility Driven Design (RDD) - methodology relying heavily on the use of responsibilities and conversations

Chapter 3 - Principles of Smalltalk

Overview

After dedicating the first two chapters to the general ideas of object-oriented programming and program development, we will now shift our attention to Smalltalk and explain its main rules and concepts including the rules of its syntax and code evaluation.

We will also introduce the use of the Inspector, the Debugger, the Workspace, and the Transcript. The purpose of the Inspector is to view the components of an object. The Debugger is used mainly to catch and trace errors in execution. It can execute Smalltalk statements message by message and provides an insight into message evaluation. The Workspace is used to test pieces of Smalltalk code before they are incorporated in methods. The Transcript is used by the environment itself to display system messages but Smalltalk programmers often use it for intermediate output during testing.

3.1. Basic rules

In this section, we will use *Farm* and *Pen* worlds to show what Smalltalk programs look like, and introduce several basic rules of Smalltalk.

Example 1. Smalltalk version of *Farm* commands

As you already know, executing an operation requires selecting a receiver and sending it a message. You have already practiced this art in *Farm 1*, *Farm 2*, and *Farm 3*. *Farm 4* shows you how you could achieve the same effect using Smalltalk – although this version of the code is still somewhat preliminary. The generated code and the use of the program's self-explanatory: Can you say which sequence of actions produced the code shown on the right hand side of Figure 3.1?

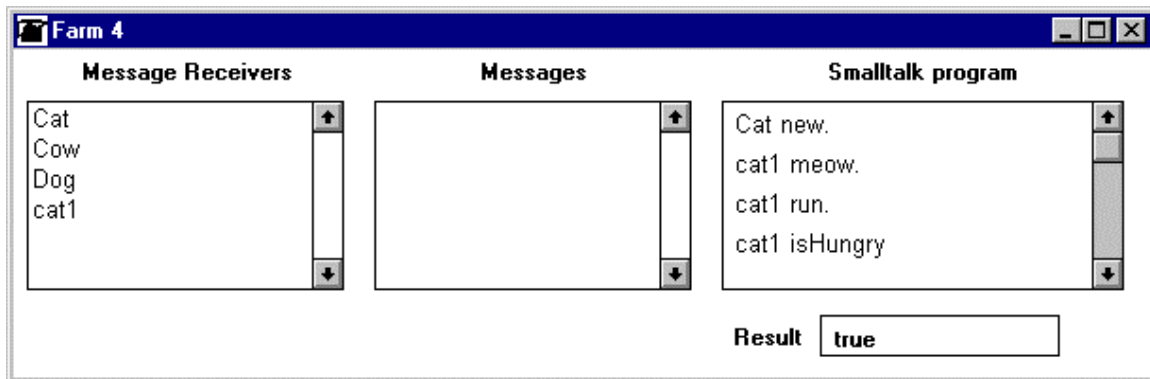


Figure 3.1. *Farm 4* displays Smalltalk equivalents of commands used in the lower level versions of *Farm*.

Pen world is another testing ground for Smalltalk code but unlike *Farm* which is too trivial, *Pen world* also allows you to solve simple problems based on drawing lines and erasing them. Figure 3.2 is an example of the *Pen world* user interface and we will immediately proceed to use it to solve a few simple tasks and examine the corresponding code. We will remind you that you should execute

PenWorld open

to run *Pen world*. In this example, we will use *Pen World 3*.

Example 2. Draw a square

Problem: Draw a 50-pixel side square whose lower right corner is the home position of the pen (Figure 3.2).

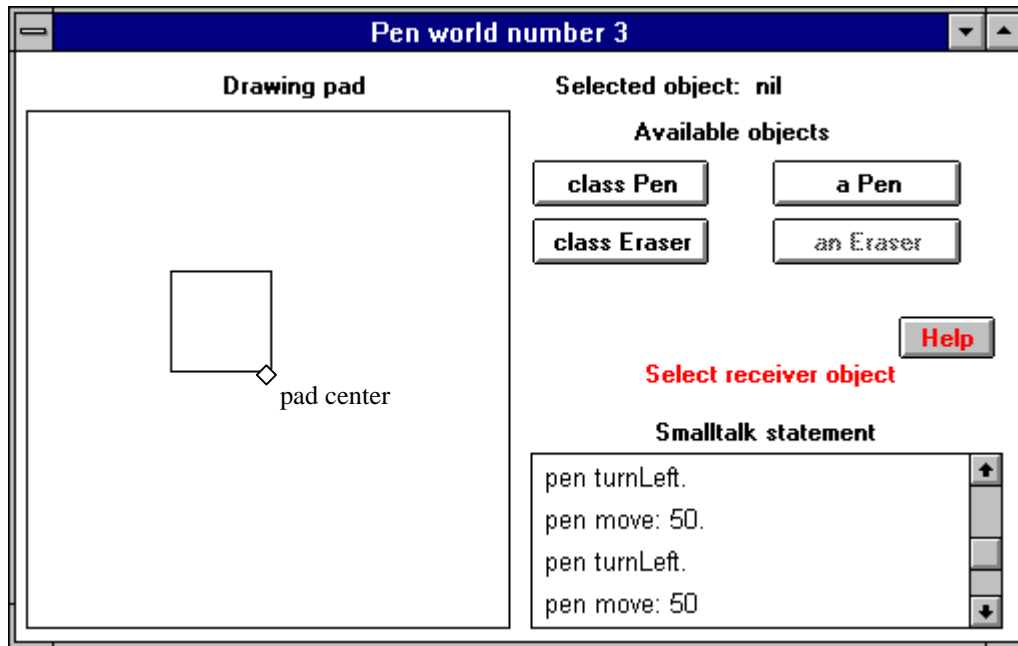


Figure 3.2. *Pen world 3* after executing Task 1. Only the last few statements are shown but the text can be scrolled to show all statements executed so far.

Solution: To be able to solve the problem, we must understand the mechanics of the *Pen World*. First, before we can use a pen or an eraser, we must create one using the *Pen* or *Eraser* class. When *Pen* creates a pen, it leaves it in the 'down' position in the center of the drawing pad and 'touching the drawing surface', ready to draw. It is oriented towards the top of the drawing pad so that when you move it, it will draw a straight line going up. The functions available through the button interface include lifting and lowering the pen, turning it left or right by 90 degrees, and moving it by a specified number of pixels (screen positions) in the current direction. Additional messages are available through the programming interface in *Pen World 5*.

As you execute the task, the text view displays a Smalltalk *statement* each time you click an *object selection - message selection* pair, and when you are finished, the text view contains the following self-explanatory code fragment:

```
Pen newBlackPen.  
pen move: 50.  
pen turnLeft.  
pen move: 50.  
pen turnLeft.  
pen move: 50.  
pen turnLeft.  
pen move: 50
```

Several notes are in order:

1. A *statement* consists of a receiver followed by a message. (This is a preliminary definition.)
2. Smalltalk programmers informally use the term *code fragment* for a sequence of Smalltalk statements written to test a programming idea. We also use code fragments to illustrate concepts and the use of existing objects, when creating new classes or methods would require too much space.

3. The term *program* is sometimes used with the same meaning as *code fragment* even though it may create the wrong impression that Smalltalk applications are big code fragments. As you know, Smalltalk applications consist of classes, with code organized into methods rather than code fragments so a code fragment is only a preliminary stage of Smalltalk programming before code crystallizes into methods. In fact, the idea of a code fragment is so foreign to Smalltalk, that before you execute one, Smalltalk converts it temporarily into a method and executes it as a message. The only real difference between a code fragment and a method therefore is that code fragments Smalltalk does not store code fragments as a part of a class. Also, whereas code fragments are usually created in a Workspace, classes are created with the browser.
4. In a sequence of statements, individual statements are separated by periods. A period is not required behind the last statement.
5. Names (*identifiers*) of Smalltalk messages and objects consist of a sequence of letters or digits and start with a letter. There is no limit on the length of an identifier and Smalltalk treats all characters in an identifier as meaningful. Examples of *legal* identifiers are aPen, PenWorld1, RedPen, aRedPenWithAThickTip, pen1, drawCircle, draw3Circles. Examples of *illegal* identifiers are three%, 3pens, this&that, and pen 1. Deviations from the naming rule may produce unexpected results. As an example, if you try to call an object red pen, Smalltalk will think that red is the receiver and pen a message.
6. Names of *classes* start with a capital letter as in Pen or Eraser; names of *instances* start with a lower case letter as in pen or eraser. Names of *all messages* should start with a lowercase letter. Smalltalk identifiers are *case sensitive* which means that two identifiers are different even if they differ only in the case of some characters. For example, redPen and redpen are different identifiers, and turnLeft and turnleft are also different. Using the wrong case happens so often that Smalltalk has a simple built-in spelling corrector: If you misspell an identifier in a program and try to execute it, Smalltalk will display the word and ask you if you want to attempt automatic correction. If you do, it will find several similar identifiers that *are* defined in the library or in your program, let you select one, and make the replacement automatically. If you misspell the case of an identifier or make a similar minor mistake, the corrector will be able to offer a solution but for more serious spelling mistakes the corrector is insufficient and you must make the correction manually.
7. To make programs easier to understand, use descriptive identifiers. As an extreme example showing the result of a very poor choice of identifiers, consider the following version of the program above in which we chose legal but meaningless identifiers:

```
XY z.  
a f: 50.  
a t.  
a f: 50.  
a t.  
a f: 50.  
a t.  
a f: 50
```

While this program would work if all the identifiers were defined, the original version gives a good idea of what is going on but this version does not give any indication of what the program does. This is an extremely important consideration because readability greatly influences code maintainability, and maintenance is the most expensive part of software development. Readability is improved by following conventions that other Smalltalk programmers use and we listed the most important ones in Appendix 5.

8. Smalltalk programmers often combine several words into a single identifier and capitalize the start of each word inside the name as in newBlackPen or turnLeft.
9. The rules of Smalltalk make it easy to understand the structure of messages. As an example, consider the statement

```
Pen newBlackPen
```

Pen is the receiver because it comes first, newBlackPen is a message. The name Pen begins with a capital letter which suggests that it refers to a class¹. This being the case, newBlackPen is a *class message*. As another example, pen in

pen move: 50

is the receiver and move: 50 is a message whose argument is 50. The name pen starts with a lowercase letter so it refers an instance of a class, and this means that move: is an instance message. The argument specifies that this move should displace the pen by 50 steps.

Main lessons learned:

- A Smalltalk statement is a receiver-message pair (preliminary definition).
- Names of objects and messages are called identifiers.
- Smalltalk rules for identifiers are as follows:
 - All identifiers must begin with a letter and the rest must consist of letters or digits.
 - Identifiers are case sensitive and have unlimited length.
 - Names of classes and class variables start with a capital letter, names of instances, instance variables, and all messages start with lowercase letters.
- A code fragment is a sequence of statements written to test a programming idea. The term program is sometimes loosely used for the same purpose although a Smalltalk program-application is a collection of classes.

Exercises

1. Write a code fragment to draw the rectangle from Example 2 in counter clockwise direction. Then execute the task in *Pen world 3* and check your answers against the automatically produced code.
2. Which of the following are not legal identifiers and why? newbook, squared, Book, aBook, new-book, newbook, new_book, negated, correct%, book@shelf, squareRoot, account, 3times, Account, factorial, display, display area.
3. Which of the legal names in the previous exercise are probably names of classes, and which ones are names of instances? Which ones are probably names of messages? Note that the only safe way to decide whether an identifier denotes a message or an object is to see it in the context of Smalltalk code.
4. Suggest descriptive Smalltalk identifiers for the following objects: a cash register class, a cash register instance, a book object in a library catalog, a green car, a passenger car class.
5. Suggest descriptive Smalltalk identifiers for the following messages: Dispense cash, check out a book, create a new object, turn left twice, close a window on the screen. See Appendix 5 for guidelines.
6. Use the System Browser to find five well selected names in the instance protocol of class Character (category Magnitude-General).
7. Is there a difference between names of class messages and names of instance messages? (Hint: Use the class setting in the Browser and check out a few messages.) If not, how can you tell whether a message is an instance message or a class message?

3.2 Maintaining access to objects - variables

If your drawing pad contains one pen and you want to send it a message, there is no ambiguity as to which pen you mean. But what if you have more pens? In a *Pen world* controlled by buttons, the solution is easy - choose the pen by clicking its image. However, if you want to control the pen by Smalltalk code, you need a different name for each pen. As we already mentioned, identifiers that denote objects are called

¹ Use this reasoning with caution since names of some other kinds of objects such as class and global variables also begin with a capital. However, in most cases a capitalized name is a class name.

variables because the state of the object attached to a variable may change during execution. In *Pen world 4*, we added variables to the displayed code and the code that *Pen world 4* produces is complete and correct Smalltalk code.

Task 2. Draw two squares

Use *Pen world 4* to draw a red square with a 70-pixel side, and a blue square with a 90-pixel side (Figure 3.3). The origin of the red square (its upper left corner) is at offset 30@50 (30 right and 50 down) from the home position at the center of the pad, the blue square's origin is the home position. Draw the squares using the pens alternately.

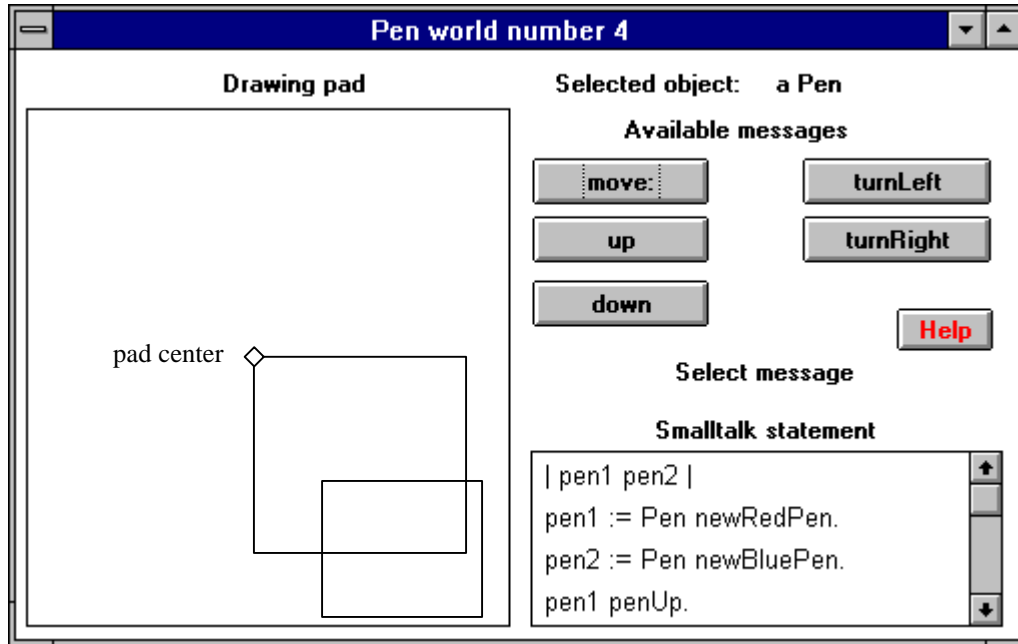


Figure 3.3. Example 2 executed in *Pen world 4*.

The algorithm to solve the problem is as follows:

1. Create two pens, one red and one blue.
2. Lift the red pen and move it to the desired origin of the red rectangle.
3. Lower the red pen.
4. Draw the rectangles using the pens alternately.

When you execute this algorithm using the pen buttons in *Pen world 4*, the text part of the window (the text subview) will display the code listed below except for the *comments* (the text surround by double quotes) that we added to relate the code to our algorithm.

```
| pen1 pen2 |  
"Create two pens"  
pen1 := Pen newRedPen.  
pen2 := Pen newBluePen.  
"Position the red pen without drawing, get it ready to draw"  
pen1 up.  
pen1 turnRight.  
pen1 move: 30.  
pen1 turnRight.  
pen1 move: 50.  
pen1 down.
```


"Draw the squares, alternating the pens"

```
pen1 move: 70.  
pen1 turnLeft.  
pen2 turnRight.  
pen2 move: 90.  
etc.
```

The first line of the code fragment is *variable declaration*. The names of all Smalltalk variables used in a method or code fragment must be declared before the first statement. As you can see, a variable declaration consists of a list of identifiers separated by blanks and surrounded by vertical bars.

If you were *writing* the program (rather than executing it by *clicking* buttons), the choice of the name would be completely up to you and you might prefer to call the pens redPen and bluePen; *Pen world 4* creates variable names automatically - and without much intelligence.

The line

"Create two pens"

is a comment. Programmers use comments for documentation and Smalltalk ignores them. A comment is any text surrounded by double quotes and you can put it anywhere you want except inside identifiers and other elementary parts of the language such as numbers. Use comments to explain the purpose of a method and to clarify less obvious code.

Immediately after variable declaration and just before the first statement, the code has not yet attached variables to any objects and its value is thus undefined. Since a Smalltalk variable must always reference an object, each variable is initially bound to a special object called nil. Object nil is the sole instance of class UndefinedObject that exists mainly for this reason.

The line

```
pen1 := Pen newRedPen
```

is called an *assignment* because it assigns an object to a variable (Figure 3.4). In this case, it binds a new blue pen to variable pen1. In the rest of the code fragment, every reference to pen1 refers to the instance of Pen created on this line and this binding persists even when the state of the pen changes.



Figure 3.4 Assignment pen1 := Pen newRedPen binds variable pen1 to a Pen object

The only way that a variable's binding can change is that another assignment binds it to a different object which is allowed but not recommended. Objects that are bound to a variable declared in a method or a code fragment remain active while the code is active whereas objects that are not bound to variables or other objects cease to be valid after they cease to be referenced. The *automatic garbage collector* of Smalltalk then automatically removes such unreachable objects from memory².

As you can see, an assignment consists of the name of a variable followed by the *assignment symbol* := (an approximation of a left pointing arrow ← to indicate the direction of the assignment), and an expression that calculates the object being assigned. Note that there is no space between : and =. The usual ways of reading an assignment such as

```
pen := Pen newBluePen
```

² Appendix 8 contains more information about garbage collection.

is *'pen gets (or refers to, is bound to, gets the value of) Pen newBluePen'*. Note that there is no space between `:` and `=`. An assignment is one of the forms of an expression and as such it returns a value – its right hand side.

After these language-related issues, note one other thing about the code: If you did not click the buttons in the same order as we did, you ended up with a program in which the same statements are executed in a different order, or maybe even a program using different statements if you used a completely different strategy. Yet, the resulting drawing could be the same. This shows that the same result can often be obtained in many different ways. Sometimes all solutions are equally satisfactory, at other times one solution may be better than another.

One should, of course, strive for the best solution but what is the best solution may depend on the context. You might expect that the best solution is the one that executes fastest or the one that requires the least computer memory. This is true in some cases but in most situations the best solution is the one that makes most sense and is reasonably efficient in terms of execution speed and use of memory. This is because the most important considerations in program development usually are how fast the application can be developed, how much it costs to develop, and how difficult it is to maintain. The cost of maintenance of a commercial product is normally higher than the cost of development and because maintainability is closely related to understandability, clarity is a paramount consideration. Even when code must run as fast as possible or use as little memory as possible, expert programmers write a readable solution first, and optimize its performance after getting the program to work, when they are certain that the overall approach is correct.

Main lessons learned:

- A variable is an identifier bound to an object. We use it to access the object in a program.
- Smalltalk recognizes temporary variables used in code fragments and methods, and instance and class variables holding an object's internal state. Temporary variables maintain access to an object within a code fragment or a method whereas instance variables are attached to an object.
- Declare a variable when you need to refer to an object several times in a code fragment or method. Objects that are not referenced several times do not need to be assigned to a variable.
- Temporary variables may also be useful to explain the logic of complex expressions.
- All temporary variables must be declared at the beginning of the code fragment or method.
- All variables are initially bound to the nil object, the single instance of class UndefinedObject.
- An assignment assigns an object to a variable. It creates a binding between the identifier and the object.
- An object remains bound to a variable until a new assignment binds the variable to another object.
- An object bound to a variable persists as long as the variable is valid: A temporary variable is valid within the code fragment or method in which it is declared, objects bound to instance variables persist as long as the object to which the variables belong, and class variables persist until the class is removed from the library.
- Unbound objects are automatically collected and discarded by the garbage collector.
- A comment is a note inserted into a program for documentation.
- Smalltalk comments consist of text preceded and followed by the double quote character.
- When writing computer programs, clarity is usually the paramount consideration.

Exercises

In the first two exercises, write an algorithm and the corresponding code on a piece of paper, execute the task by clicking buttons in the *Pen world*, and check whether the generated code is the same as yours.

1. Draw letter S in the center of the drawing area of *Pen world 4*. The letter should be red, 50 pixels high, and 20 pixels wide.
2. Draw letter X in the center of the drawing area of *Pen world 4*. The letter should be green, 50 pixels high, and 20 pixels wide.
3. Try the more advanced *Farm* worlds. They use similar formats as *Pen* worlds.

3.3 Writing and executing programs

Up to this point, we have been ‘programming’ with buttons. In this section, we will switch to real programming using *Pen world 5* (Figure 3.5) which provides the same functionality as regular Smalltalk programming tools such as the Workspace.

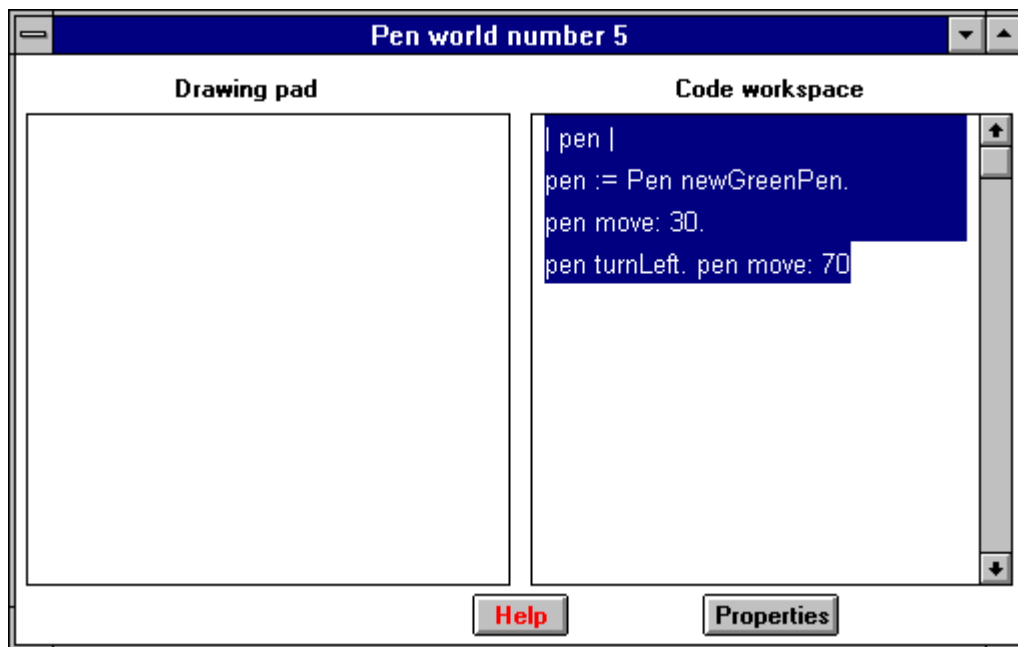


Figure 3.5. *Pen world 5* with code selected for execution.

To write and execute a code fragment in *Pen world 5*, type the code into the text view, select it, and execute it with the *do it* command from the <operate> menu³. We leave it to you to write and execute some of the exercises at the end of this section. In all cases, write the code on a piece of paper first, enter it very carefully into the text view, and try to execute it. If you have problems, read the rest of this section to see how to recover. If you can’t solve the problem, go to *Pen world 4*, execute the task using buttons, study the automatically created code, correct your program in *Pen world 5* accordingly, and try again.

³ Since we have introduced two buttons so far - the <select> and the <operate> buttons, let’s introduce the third one: If you have a three-button mouse, Smalltalk’s name for the right mouse button is <window> button because it opens the <window> menu with window-related commands. If your mouse has only two buttons, pressing the <Ctrl> key and the right button has the same effect as pressing the rightmost key on a three-button mouse. In the first implementations of Smalltalk, the left, middle, and right buttons were called *red*, *yellow*, and *blue* and some important Smalltalk code still uses these names.

Dealing with simpler errors

Sooner or later, you will type and attempt to execute incorrect code. Code can be incorrect in more than one way:

- It may violate some rules of the language - the *syntax* rules⁴. As an example, you may have used illegal characters in an identifier name or typed a space between : and = in the assignment. If you enter syntactically incorrect code, the compiler will notify you and Smalltalk will not even start executing the code. Once you master the simple Smalltalk syntax, correcting syntax errors is easy.
- A syntactically correct code may use a meaningless identifier such as a misspelled or undeclared variable or class name, or a message name whose declaration does not exist. The compiler will again notify you and correcting the mistake should be easy, especially because the corrector can help you find correct spelling and the environment can declare variables for you.
- A frequent cause of errors is sending a legal message to the wrong object. As an example, even though the following expression contains a legal receiver and a legal message,

`'abc' squared`

is illegal because string objects don't understand the `squared` message (you cannot square a string.) This kind of error is usually easy to correct. When you try to execute code with this problem, Smalltalk will execute it up to the point where it breaks and open an Exception window. The use of Exception windows is explained below.

- The really unpleasant error is a *logic error*. In this case, everything is legal and executes but it does not produce the correct result. As an example, if you are trying to draw a square and turn the pen left instead of right, you obtain a drawing but not the one you wanted - this is an error in the logic of the program. If you have a logic error (your program works but produces an incorrect result), read your code very carefully and try to correct it. If this does not help, use the Debugger as explained below.

The first two kinds of errors listed above occur at *compilation time* (when the code is translated for execution and before it can be executed), the last two occur at *run time* (when encountered during the execution of the code). We will now illustrate some of the possible problems in more detail.

Misspelled name

If you misspell the name of a variable or a message, Smalltalk opens a Notifier window as in Figure 3.6. In this case, we incorrectly typed the name of a message as `newredpen` instead of `newRedPen`. In a minor misspelling such as this one, Smalltalk will be able to find the correct name if you click *correct it*. If the mistake is bigger, *correct it* will not be able to help. In this case, select *abort*, return to your program, and correct it manually.



Figure 3.6. Error window indicating that code contains an unrecognizable name (here `newredpen`).

⁴ See Appendix 7 for a complete listing of Smalltalk syntax.

Receiver does not understand message

If everything is typed correctly but the receiver does not understand the message because its class does not define it and does not inherit it, Smalltalk opens an Exception window such as the one in Figure 3.7. This Exception window popped up when we tried to execute

'75' squared

The message displayed in the window clearly identifies the problem (if you know that ByteString refers to strings) and we can thus close the Exception window by clicking *Terminate*, correct the expression to

75 squared

and re-execute the code with better success. If the problem is not clear, click *Debug* and open the Debugger. We will talk about the Debugger shortly.

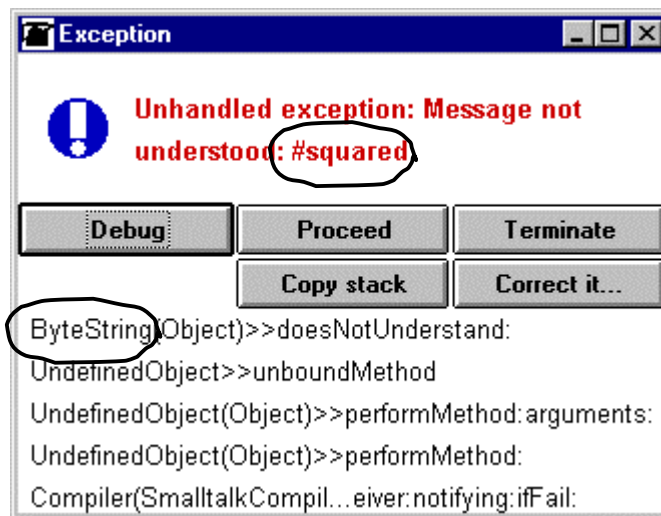


Figure 3.7. Exception window indicating that the receiver (a ByteString object) does not understand message squared. Click *Terminate*, return to your program, and correct the error.

Main lessons learned:

- On a three-button mouse, the right button is the <window> button. Use it to activate the <window> menu with window-related commands. If you only have two buttons, the right button in combination with <Ctrl> has the same effect.
- The <select>, <operate>, and <window> buttons were originally called *red*, *yellow*, and *blue* and some Smalltalk methods still use these names.
- The rules of a language are called syntax rules, and code that violates a syntax rule contains a syntax error.
- Besides syntax errors, code can contain run time errors such as messages sent to receivers that don't understand them, and logic errors.
- Smalltalk catches all but logic errors and opens notifier or exception windows explaining their nature.

Exercises

1. Write an algorithm and a program to draw two squares centered at the home position. The inner square should be red with size 40 pixels, the outer square should be blue with a side of 60 pixels.

2. Objects in *Pen World 5* can execute messages that we have not listed so far. As an example, you can change the width of the tip of the pen, its drawing speed and its color, and you can perform additional operations with the eraser as well. Use the System Browser to find, list, and explain all pen and eraser messages, formulate several problems using them, program them, and test them.
3. Open a Workspace and enter and execute the following code fragment. Be careful to type it correctly.

```
| number string |  
"Get a number from the user."  
string := Dialog request: 'Enter a number' initialAnswer: '0'.  
number := string asNumber.  
"Calculate and display its square in the Transcript."  
Transcript cr.  
Transcript show: 'The square of '.  
Transcript show: number printString.  
Transcript show: ' is '.  
Transcript show: number squared printString
```

The program requests a number from the user, calculates its square, converts it to a string, and displays it in the Transcript. Identify all parts including the declaration, variables, assignments, receivers, messages, statements, and comments.

3.4 More about variables

We have, by now, encountered the term *variable* in several contexts and it is time to summarize and refine our definitions:

- A variable is an identifier bound to an object. Sending messages to a variable is equivalent to sending messages to the object referenced by the variable.
- Classes keep their properties in *class variables*. Class variables are listed in the definition of the class and their value is shared by the class and its subclasses. They can be accessed by the class, its subclasses, and *their instances*. The lifetime of objects bound to class variables is the same as the lifetime of the class and since classes are persistent (saved in the 'image' file⁵ when you execute the *save as* or *exit* with *save and exit* command from the launcher window – and restored when you start Smalltalk), they last until the class defining them is deleted from the library.
- Instances keep their values in *instance variables*. Instance variables are private to the instance which means that they cannot be directly accessed by any other object. In particular, instance variables cannot be accessed by the class. Other objects can access an instance variable only if the class definition provides a method that accesses it. You will find all instance variables in the definition of the class and the lifetime of objects bound to instance variables is related to the lifetime of the instance: When an object ceases to exist, the objects bound to its instance variables are discarded and collected by the garbage collector *unless* they are also bound to some other variables or objects that are still 'live'.
- Variables declared in methods are called *temporary variables*. They are valid only within the method and their validity is limited to the method. When the execution of a method ends, objects bound to its temporary variables are discarded unless they are bound to some surviving objects.
- When a temporary variable is first encountered in a program or when an instance variable becomes valid when an instance is created, it is automatically bound to nil and new values can be assigned to it with the assignment expression. This assignment may happen only in the appropriate context:
 - Class variables may only be assigned in class or instance methods.
 - Instance variables may only be assigned in instance methods.
 - Temporary variables may only be assigned in the method in which they are declared.

⁵ The image file is one of the essential components of the Smalltalk environment. See Appendix 8 for more about it.

The range of code in which an identifier is valid is called its *scope*. The scope of a temporary variable is thus the method or code fragment in which it is declared. The scope of an instance variable consists of the instance methods in the class in which the variable is declared and in all its subclasses. The scope of a class variable consists of the class-, subclass-, and instance methods.

Type of variable	Accessible from (scope)
temporary	method where declared
instance	instance method of class where defined and its subclasses
class	class and instance method of class where defined and its subclasses

To illustrate the concepts of instance and class variables, Figure 3.8 shows the definition of class `Date`. It shows that each instance of `Date` has two instance variables, one called `day` and one called `year`. The class also has five class variables and inherits several variables from superclasses. (The easiest way to see all class variables defined for a class is to use the *class var refs* in the <operate> menu of the class list or use the Full Browser in Advanced Tools.) `Date` does not inherit any instance variables. (Use *inst var refs* in the <operate> menu.)

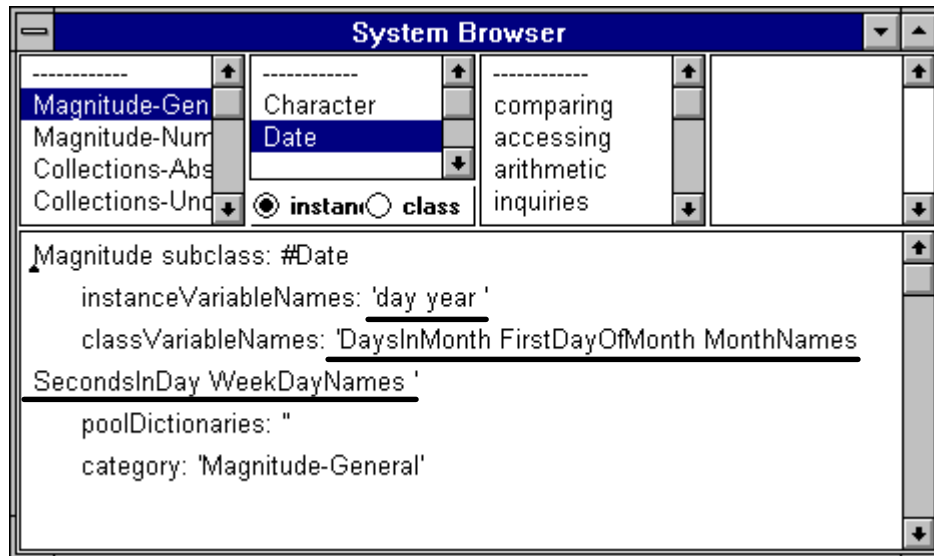


Figure 3.8. Definition of class and instance variables in class `Date`.

Inspecting variables

To see the components of an object, send it the message `inspect`. This will open an *Inspector* window. As an example, when you enter

`(Date today) inspect`

into a Workspace and execute it with *do it*, you will get an Inspector on the object returned by `Date today` (Figure 3.9). We used the parentheses only to indicate the logic of the expression (we are inspecting the object `Date today`) but we can get the same result by executing

`Date today inspect`

or simply by executing

`Date today`

with the *inspect* command instead of using the inspect message and *do it*.

The *self* item at the top of the list refers to the receiver of the inspect message, in this case the *Date* object. The remaining items provide access to all instance variables of the inspected object, both inherited and defined in its class.

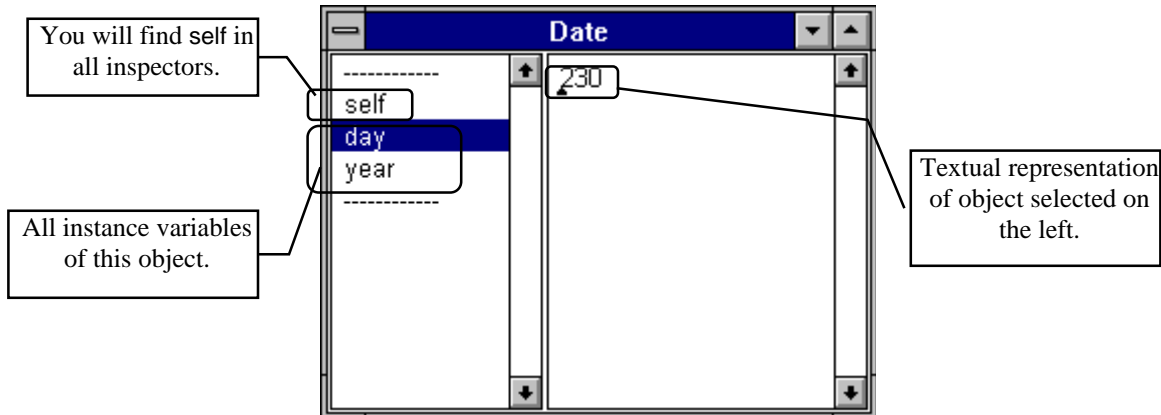


Figure 3.9. Inspector on the *Date* object created by *Date today*.

Class *Date* provides a good example of the need for class variables (Figure 3.10). Its class variable *FirstDayOfMonth* contains the number of the first day of each month in a year counted from the start of the year, *MonthNames* contains the names of all months, and so on. These objects are of interest to many class and instance methods in *Date* and this programs and is why they are stored in class variables

Besides viewing the values of an object's instance variables, the inspector can also be used to inspect the variables themselves. As an example, if you selected *day* as in Figure 3.9, executing *inspect* in the inspector's <operate> menu will open an inspector on the *day* object, and so on. Inspecting objects to any desired depth is one of the most common activities during program testing.

Since classes are also objects, we can inspect them too. To inspect *class Date*, enter

Date inspect

into a Workspace and execute it with *do it*, or execute *inspect* over the name of the class. This will open the inspector in Figure 3.10. Most of the items in the list don't make much sense at this point because classes are rather special objects, but *classPool* contains all class variables with their values. Another interesting item is variable *methodDict* which contains the list of all instance methods defined in this class.

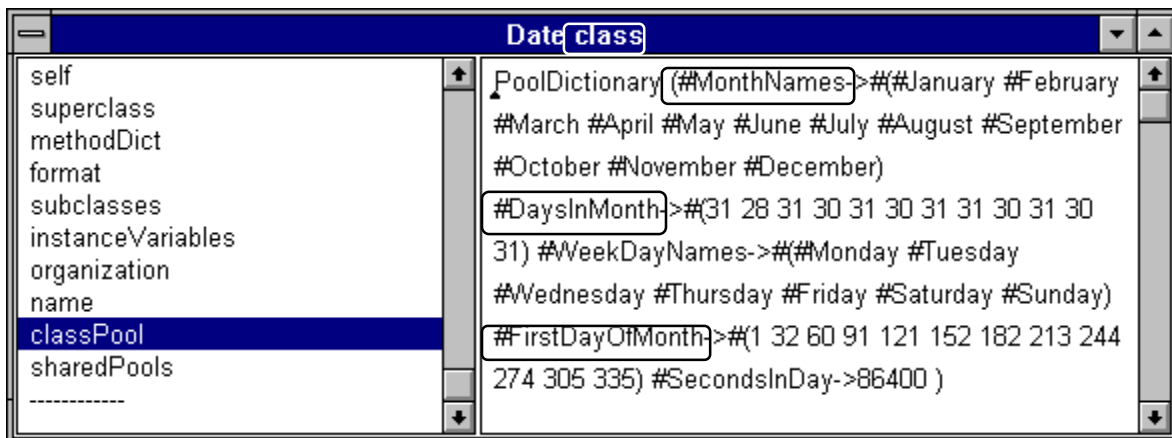


Figure 3.10. Inspector of class *Date*. The window label says that the inspected object is a class. Class variables are highlighted on the right side.

Main lessons learned:

- The scope of a variable is the range of code in which the variable can be directly used or assigned to.
- The scope of a temporary variable is the method in which it is defined, the scope of an instance variable are instance methods of the class and its subclasses, and the scope of a class variable are class and instance methods of the class and its subclasses.
- The lifetime of an object is the time during which it is valid. Except for certain base objects, an object is live when other live objects refer to it.
- Values of instance variables of an object can be inspected by an Inspector.
- To view class variables, inspect the class object itself and then inspect its classPool variable.
- Smalltalk uses the special identifier `self` to refer to the receiver of a message.

Exercises

1. Create the following objects and inspect them and their components:
 - a. An instance of Time obtained with message Time now.
 - b. An instance of Rectangle obtained with message Rectangle origin: 12@13 corner: 150@300.
 - c. A Rectangle obtained with Rectangle origin: 12@13 extent: 300@400.
2. Inspect the class variables of the following classes: Point, Rectangle, Random, Time, Float. Try to explain their meaning and justify the need for them. (Hint: Read also the class comment.)
3. Examine ten randomly selected classes using the System Browser and count how many of them have instance variables and how many have class variables. What do you conclude about the relative frequency of instance variables and class variables defined in a class?

3.5 Smalltalk messages

After examining object properties (class and instance variables) we will now turn to messages. As you have already seen, Smalltalk recognizes three types of messages - *unary*, *binary*, and *keyword* messages and each of them returns a single object as its result. We will now outline the rules for forming their names (*selectors*) and then give more detailed explanations accompanied by several examples.

- A *unary message* consists of a single identifier such as `turnLeft`. It does not have any arguments.
- Selectors of *binary messages* use special symbols such as `+` and `@` and have exactly one argument. As an example, `+` is a binary message used as in `3 + 4`, and `@` is used to create Point objects as in `13@45`.
- The selector of a *keyword message* consists of one or more keywords and each keyword is an identifier followed by a colon. Each keyword is followed by an argument as in `move: 70`. Note that the colon immediately follows the text without any intervening blanks.

All three types of messages may be used for either class or instance messages. The built-in library contains many unary, binary, and keyword methods and you can define your own messages of any type as well, thus creating your own 'vocabulary'. Smalltalk treats built-in and user-defined messages exactly the same way. We will now show some examples of all three kinds of messages and we encourage you to execute them and inspect their results.

Unary messages

We have already encountered several examples of unary messages including

Pen newBluePen	"A class message (receiver is <i>class</i>) Pen which creates a new pen."
CatFactory new	"A class message which creates a new cat."
cat meow	"An instance message (receiver is an <i>instance of class</i> Cat)."
pen turnLeft	"An instance message."

Time now	"A <i>class</i> message that returns a Time object for the current time."
cow isHungry	"An instance message."

The following are several more rather self-explanatory examples:

3.14 negated	" negated is an instance message that changes the sign of a number."
Random new	" new is a <i>class</i> message used by most classes to create a new instance."
100 factorial	" factorial is an instance message, here calculating 100*99*98*97* ... *3*2*1."
0.3 cos	" cos is an instance message that calculates cos(x), the cosine of its receiver."
25 squared	" squared returns the square of its receiver."
'Abc' asUppercase	" asUppercase converts its receiver to uppercase letters, here 'ABC'."
3.14 asFraction	" asFraction is an instance message that converts a number to a Fraction."

Binary messages

Binary messages are used mainly for arithmetic operations and for very common one-argument messages. A binary selector can be formed by one or more symbols⁶ selected from a variety of characters including + - , / \ * & < > = and @. Several other characters such as ! and ? are also allowed but never used.

The following are several examples of binary messages. Type them into a Workspace and execute them with *inspect* or *print it* to see the result.

100 @ -340	"Message to 100 - creates a new Point object."
3 + 17	"This and the following six binary messages do arithmetic."
5 - 175	
56 * 43	
187 / 23	"This and the following three messages perform various division operations."
187 // 23	
187 \ 23	
187 \ 23	
'the name of the file is', ' ass.12'	"Combines (concatenates) two strings into one."
(12@15) + (30@40)	"Adds together two points - vector arithmetic."
3 > 8	"Returns the false object because it is not true that 3 is greater than 8."
3 <= 5	"Returns the true object because 3 is less or equal to 5."

All these messages are instance messages as their receivers are instances of numbers or strings. There are no binary class messages but there is no rule against defining one.

Keyword messages

Keyword messages are the essence of Smalltalk's expressive power because they let you create methods with any number of arguments. In fact, Smalltalk could exist without unary messages (one could use keyword messages and ignore arguments) and without binary messages (they could be implemented as keyword messages with one keyword). One could say that unary and binary messages are included just for convenience and readability.

The following are a few examples of keyword messages that you should try out:

Keyword messages with a *single keyword*:

Dialog confirm : 'Delete file?'	"Class message; opens dialog window, returns true or false."
Dialog warn : 'The disk is now full.'	"Opens a warning window with an OK button."
'abcdef' isLike : 'abc*'	"Returns true - checks <i>wild card character</i> * matches."
'aVeryLongFilename' chopTo : 8	"Returns 'aVername' - original chopped to 8 characters."

⁶ The traditional rules allow only one or two special characters and impose special rules on the use of the – character. The current Smalltalk Standard Draft proposal removes most of these restrictions.

'a cat' **spellAgainst:** 'a car'

"Returns 80 - degree of similarity between the two strings."

Whereas one-keyword messages don't seem to cause any problems, keyword messages with multiple keywords may be more difficult to read. Beginners often don't understand that the multiple keywords form a single message. The following are a few examples of multiple-keyword messages, one message per line.

Keyword messages with *two keywords*:

Dialog **request:** 'Name' **initialAnswer:** 'John'
Rectangle **origin:** 10@20 **corner:** 100@200
3 **between:** 10 **and:** 20

"Opens dialog, returns string entered by user."
"Class message creating a rectangle."
"Returns false: 3 is not between 10 and 20."

Keyword message with *three keywords*:

'Smalltalk' changeFrom: 1 to: 5 with: 'Big'

"Replaces the first five characters, giving Bigtalk."

Keyword messages with more keywords are allowed but are not very common.

In closing, let us note that the name of a message excluding arguments is called the *message selector*. As an example, `between:and:` is the selector of message `between: 5 and 19`; `+` is another selector and `3 + 5` is an expression using it.

Main lessons learned:

- Smalltalk recognizes unary, binary, and keyword messages.
- Unary messages consist of an identifier only and do not have an argument
- Binary messages are formed by combining one or two special symbols from a set of characters including `+` `-` `/` `*` `&` `@` `,` `<` `>` `=`. A binary message has exactly one argument.
- Keyword messages are a sequence of one or more identifiers followed by a colon and an argument.
- A keyword is a selector followed by a colon.
- User-defined and built-in messages are treated in the same way.
- The name of a message (without its arguments) is called the message selector.

Exercises

1. Execute the code fragments from this section.
2. Examine 10 randomly selected method protocols using the Browser and estimate the relative frequency of unary, binary, and keyword messages.

3.6 Combining messages

Each message returns an object and since this object can be used as the receiver of another message or as an argument, messages can be combined or *chained*. As an example, if you want to add together two points, you can write either

```
| point point1 point2 |  
point1 := 12 @ 15.  
point2 := 37 @ -81.  
point := point1 + point2.  
etc.
```

or simply

```
| point |  
point := (12 @ 15) + (37 @ -81).  
etc.
```

if you don't need the intermediate points. If you do need the two intermediate points, you can also write

```
| point point1 point2 |  
point := (point1 := 12 @ 15) + (point2 := 37 @ -81 ).  
etc.
```

because an assignment returns the value of its right hand side.

Execute all three forms with *print it* or *inspect* in a Workspace to see that you will get exactly the same result. As another example, to test whether $12^{15}/5$ is less than 20 factorial, you could execute

```
| number1 number2 |  
number1 := 12 raisedTo: 15.  
number1 := number1 / 5.  
number2 := 20 factorial.           "20*19*18*17*...*4*3*2*1."  
number1 < number2                 "Returns true or false. "
```

or simply

```
((12 raisedTo: 15) / 5) < (20 factorial)
```

Execute both fragments to see that they both return the same result.

All experienced Smalltalk programmers combine messages to simplify code and to eliminate unnecessary variables. However, don't overuse this feature because expressions can become too complicated and difficult to understand, possibly producing different results than anticipated.

Obviously, if you want to combine messages and get the correct result, you must know how Smalltalk interprets combined messages. This will be explained in the next section.

Main lessons learned:

- Each Smalltalk message returns an object and this object can be used as an argument or as the receiver of another message. Combining (chaining, nesting) messages in this way makes code more compact.
- Chaining is common but use it with moderation to keep the code readable.

Exercises

1. Read the following expressions and classify them as unary, binary, and keyword messages. Try to predict their meaning and execute them.
 - a. $3 + 2$ factorial
 - b. $(3 + 2)$ factorial
 - c. $0.3 \sin^2 + 0.3 \cos^2$
 - d. $(0.3 \sin^2) + (0.3 \cos^2)$
 - e. 15 factorial between: $(15 \text{ raisedTo: } 7)$ and: $(15 \text{ raisedTo: } 22)$
 - f. $12@15$ corner: $123@75$ "Creates a rectangle using two point objects."
 - g. $(\text{Point } x: 27 * 0.7 \cos y: 45 * 0.7 \sin) < 45@34$ "Is Point on left above and to the right of the Point on the right?"
2. Imagine two solutions to the same problem. The first solution stores a large object in a variable, the second avoids the need for a variable by combining messages into larger expressions. Explain the possible advantage of the first approach.
3. What do you think is the effect of the following expressions. Test your prediction.
 - a. $(3 + 5) / 2$ "For integer receiver and argument, message / returns a fraction."
 - b. Rectangle origin: $((12@15) + (17@18))$ extent: $(90@x5 * 3)$
 - c. Dialog request: ('Enter name of ', string) initialAnswer: ('file.', extension) assuming that the value of variable string is 'the file', and that extension is '*'.
3. Explain the differences between the following expressions:
 - a. $x := 'abc'$ asUppercase
 - b. $(x := 'abc')$ asUppercase

- c. `y := (x := 'abc') asUppercase`
- 4. Write Smalltalk expressions to solve the following problems and test that they calculate the correct result by typing your code into the Workspace and evaluating it with *print it*.
 - a. The hypotenuse and one side of a right-angle triangle are 5 and 4 points long respectively. Calculate the third side. (Hint: There are two messages to calculate the square. One is `**`, the other is `squared`. The message to calculate the square root is `sqrt`.)
 - b. What is the angle between the two given sides in the previous example? (Hint: Look in class `Number` for the required messages.)
 - c. In a certain country, sales tax consists of a part that goes to the federal government and a part that goes to the provincial government. The first tax is 7 percent of the price, the second tax is 8 percent of the result of adding the first tax to the price. Write an expression to calculate the total amount payable for an item whose price is \$20.
 - d. A triangle has sides that are 45 and 30 units long. The angle between the two sides is 45 degrees. Calculate the length of the third side. (Hint: Trigonometric functions use radians rather than degrees. Use message `degreesToRadians` to perform the conversion.)

3.7 Nesting of Expressions

Without knowing how Smalltalk interprets combined messages, we cannot know what a combined message calculates. Fortunately, the rules are very simple and usually correspond to the order in which expressions are evaluated. The formal definition of the rules constitutes the syntax of Smalltalk (Appendix 7) but since their interpretation by a beginner would not be easy, we give the following informal description of expression evaluation:

1. All expressions in *parentheses* are evaluated first, from left to right. This evaluation uses steps 1 to 4 again as necessary.
2. All *unary* messages are evaluated next, from left to right.
3. All *binary* messages are evaluated next, from left to right.
4. The remaining *keyword* message, if any, is evaluated.

To clarify these simple but rather abstract rules, we will now give several examples of Smalltalk expressions and evaluate them step by step. Each example adds a new feature to the discussion. Our detailed analysis may make execution appear complicated but this is because we are tracing the rules in a machine-like fashion. If you follow the rules in a more humane way, the execution will appear much simpler.

Example 1

Consider the following statement, presumably a part of a larger code fragment:

```
...  
3 factorial raisedTo: 3.  
...
```

This statement is executed as follows:

- Step 1. *Parenthesized messages*: There are no parentheses so nothing happens in this step.
- Step 2. *Unary messages*: `factorial` is a unary message. How do we know that? The first item in the statement - number 3 - must be the receiver because every expression begins with a receiver; consequently, `factorial` must be a message. It cannot be a binary message (these use special characters), and since it is not followed by a colon, it is not a keyword message. So it must be a unary message.
- As a consequence, `3 factorial` is evaluated first, giving $3 \times 2 \times 1 = 6$ and the original expression can now be thought of as

6 raisedTo: 3

There are no more unary messages in this statement so we are finished with Step 2.

Step 3. *Binary messages*: None in this statement (no special symbols).

Step 4. *Keyword message*: Message raisedTo: is a keyword message (it follows a receiver and consists of an identifier followed by a colon) with argument 3. Executing it returns $6^3 = 216$.

Note that the expression evaluates exactly as

(3 factorial) raisedTo: 3

Example 2

The following statement attempts to test whether 12^4 is between 7^5 and 10^5 and store the result in temporary variable condition:

```
| condition ... |  
...  
condition := 12 raisedTo: 4 between: 7 raisedTo: 5 and: 10 raisedTo: 5  
...
```

Unfortunately, this statement will not even start executing because we did not write it properly. When you examine what we wrote, you will see that 12 is the receiver, so raisedTo: must be a keyword. Number 4 is its argument and between: is another keyword. Continuing in this way, we find that we are telling Smalltalk to execute a keyword message with selector raisedTo:between:raisedTo:and:raisedTo:. Such a message is not in the library and Smalltalk will not accept the code. This is an example of a statement that is syntactically correct but invalid because its interpretation implies a non-existent message. Smalltalk will catch this error and display the dialog in Figure 3.11.

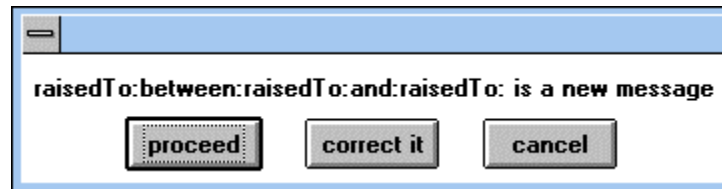


Figure 3.11. Error produced when trying to execute the first version of Example 2.

Note that the result of sloppy writing could be much worse: If we wrote the expression in such a way that its interpretation does give a message which is in the library, Smalltalk would execute it even if the message was different from the one that we had in mind. This shows that it is important to be very careful when combining messages and to make sure that the code is easily readable. We will now rewrite the assignment to produce the desired effect.

In essence, we are trying to determine whether a certain number is between two other numbers. We will thus put parentheses around the expressions that calculate the three numbers and use them as the receiver and the two arguments of between:and: respectively as follows:

```
| condition ... |  
...  
condition := (12 raisedTo: 4) between: (7 raisedTo: 5) and: (10 raisedTo: 5)  
...
```

Evaluation of this statement proceeds as follows:

1. *Parenthesized expressions* from left to right. Parenthesized expression

12 raisedTo: 4

is evaluated by re-applying evaluation rules:

1. *Parentheses*: Expression 12 raisedTo: 4 does not contain any parenthesized expressions.
2. *Unary*: It does not contain any unary messages.
3. *Binary*: It does not contain any binary messages.
4. It contains the *keyword* message raisedTo: and when we evaluate it, we get 20736. This reduces the original expression to

20736 **between:** (7 raisedTo: 5) **and:** (10 raisedTo: 5)

We are still in Step 1 of the original evaluation and we thus proceed scanning the original statement looking for more parenthesized expressions. We now find the parenthesized expression

(7 raisedTo: 5)

and evaluate it just as the previous parenthesized expressions, getting 16806. The original expression has now been reduced to

20736 **between:** 16806 **and:** (10 raisedTo: 5)

We are still not finished with Step 1, and proceeding to the right we find the last parenthesized expression

(10 raisedTo: 5)

evaluate it, and obtain 100000. The original statement now effectively reduces to

condition := 20736 between: 16806 and: 100000

2. This form of the original expression does not contain any *unary* messages.
3. There are no *binary* messages.
4. The expression contains the *keyword* message between:and: with receiver 20736 and arguments 16806 and 100000. Evaluation returns object true because it is true that the value of the receiver is between the values of the two arguments.
5. Object true, the result of the last step, is assigned to variable condition and execution proceeds to the next statement in the program.

Example 3

Using evaluation rules, you can find that

3 + 7 * 3

is evaluated as

(3 + 7) * 3 = 10 * 3 = 30

because Smalltalk evaluates from left to right with no exceptions. In conventional arithmetic, multiplication has precedence over addition and the result would be the same as for

3 + (7 * 3) = 24

If this is what you intended, you must use parentheses as in

$(3 + 7) * 3$

Parenthesizing arithmetic expressions is highly advisable even when it is not required because misinterpreting arithmetic expressions is a very frequent error in Smalltalk programs. Don't worry about making your programs less efficient by using unnecessary parentheses – the compiler will ignore unnecessary parentheses when it creates the executable code.

Example 4

In a fictitious program to calculate tax using notoriously obscure tax rules, we need to add the sum of variables `income1` and `income2` multiplied by the weighting factor 0.13 to `income3` multiplied by 0.17, and subtract the sum of variables `expenditure1` and `expenditure2` weighted by 0.06. The result is stored in variable `tax`.

One way to solve this problem is to declare a variable for each component of the calculation and then do the final calculation on these variables as follows:

```
| part1 part2 part3 income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2 := 700.
part1 := (income1 + income2) * 0.13.
part2 := income3 * 0.17.
part3 := (expenditure1 + expenditure2) * 0.06.
tax := part1 + part2 - part3.
...
```

This program directly reflects the original formulation and we are thus quite confident that it works correctly. However, since the calculation is not all that complicated, we decide to eliminate the auxiliary variables and parenthesize each part of the statement to capture the calculations described in the specification. In a way, the parentheses replace the auxiliary variables which we don't need in the long term anyway. This gives the following formulation:

```
| income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2 := 700.
tax := (((income1 + income2) * 0.13) + (income3 * 0.17)) - ((expenditure1 + expenditure2) * 0.06).
...
```

Evaluation of the last statement proceeds as follows:

1. *Parenthesized* expressions from left to right. The first parenthesized expression is

$(income1 + income2) * 0.13$

because we read the parentheses from the outside, and to evaluate it we proceed as follows:

1. Evaluate the first parenthesized expression

`income1 + income2`

by applying the five steps again:

1. No *parenthesized* expressions.
2. No *unary* messages.
3. We have *binary* message +. Smalltalk sends + to the object referenced by income1 and checks gets the result 3000. Steps 4 and 5 don't apply and we return to the original level of evaluation.

Expression

(income1 + income2) * 0.13

has now been reduced to 3000 * 0.13.

There are no more parenthesized expressions at this level and we continue to the next rule.

2. There are no unary *messages* in 3000 * 0.13.
3. Expression 3000 * 0.13 contains the *binary* message * and evaluates to 390. This ends the evaluation of this part of the statement (no keyword messages, no assignment) and we return to the original statement which is now effectively reduced to

tax := 390 + (income3 * 0.17) - ((expenditure1 + expenditure2) * 0.06).

We leave it to you to evaluate the rest of the statement and to check that the result is the same as the result of the formulation using temporary variables. To do this, execute and inspect both versions in the Workspace.

Main lessons learned:

- Smalltalk uses the following rules to evaluate expressions:
 1. Evaluate all parenthesized expressions from left to right applying rules 1 to 4 to their contents.
 2. Evaluate all unary messages from left to right.
 3. Evaluate all binary messages from left to right.
 4. Evaluate the resulting keyword message, if any.
- In applying evaluation rules, Smalltalk does not make any exceptions. In particular, it does not give precedence to any arithmetic binary messages.
- Combining expressions makes code more compact and eliminates variables. It may, however, make code difficult to understand and should not be overused.
- If an expression becomes difficult to read, divide it into consecutive statements and use temporary variables or parenthesize it.
- Be very careful with arithmetic expressions - conventional arithmetic operators use precedence but Smalltalk does not.
- Be careful with keyword message, especially when their arguments are calculated. They can be hard to understand and write correctly.

Exercises

1. Check the correctness of the syntax of the following expressions, identify receivers, arguments, and messages, and convert them to more readable but equivalent forms where appropriate. Interpret their meaning and execute them where possible.
 - a. 3 + 4 * x // 37 - 5
 - b. self account add: 1.4 * bill to: (transactions invoiceFor: currentTransaction)
 - c. (5 raisedTo: 3 + 7*x) gcd: 5 factorial + x
2. Explain the order of evaluation of the following expressions and represent it by bracketing. Execute them with *print it* to confirm your analysis.
 - a. 3 + 2 factorial
 - b. (3 + 2) factorial
 - c. 0.3 sin squared + 0.3 cos squared
 - d. (0.3 sin squared) + (0.3 cos squared)
 - e. 15 factorial between: (15 raisedTo: 7) and: (15 raisedTo: 22)

- f. 12@15 corner: 123@75
- g. (Point x: 27 * 0.7 cos y: 45 * 0.7 sin) < 45@34
- 5. Evaluate the following expressions and test the correctness of your analysis by evaluating the expressions in a Workspace.
 - a. (3 + 4) factorial / 2 squared
 - b. 0.7 sin between: 0.4 sin and: (0.1 cos) squared
 - c. (Rectangle origin: (10@30) corner: (100@70)) intersects: (Rectangle origin: (75@97) extent: (30@40))
- 6. Can any of the parentheses in the above exercises can be deleted without affecting the result?
- 7. Examine 20 randomly selected method definitions and find the
 - a. average number of lines in a method.
 - b. average number of statements per method.
 - c. percentage of statements using combined messages.
 - d. average number of messages per statement.
- 8. Check what happens when we modify the last statement of the tax example as follows:
| income1 income2 income3 expenditure1 expenditure2 ...|
income1 := 1000.
income2 := 2000.
income3 := 100.
expenditure1 := 500.
expenditure2 := 700.
tax := income1 + income2 * 0.13 + income3 * 0.17 - expenditure1 + expenditure2 * 0.06.
The syntax of the last statement is correct, the message sends are all legal because the messages are understood by their receiver, but the logic implied by the order of evaluation is wrong and so the result is also wrong.

3.8 Tracing message evaluation with the Debugger

In the previous section, we introduced message evaluation rules and demonstrated them by manual evaluation. We will now show how you can use the Debugger to do this for you.

Example

We want to execute the following code fragment step-by-step using the Debugger:

```
| x |  
x := 7.  
x := 3 factorial raisedTo: 3 + x.
```

To trace execution with the Debugger, we will add a halt message at the point where we want to start tracing, in our case just before the first assignment:

```
| x |  
self halt.                                "Open an Exception window to allow continuation in the Debugger."  
x := 7.  
x := 3 factorial raisedTo: 3 + x
```

The halt message opens an Exception window which allows you to open the Debugger to trace the execution of the program and inspect its components. Since halt breaks the execution of the program, inserting a halt message is usually referred to as *setting a breakpoint*. The halt message is defined in class Object and it is thus inherited by every class and understood by all objects. Smalltalk programmers usually send halt to self - the receiver of the message that is currently executing. A code fragment is treated as an unboundMethod message sent to nil, and self in a code fragment thus refers to nil.

Enter the program into a Workspace and execute it with *do it*. When execution starts, it immediately opens the Exception window in Figure 3.12. The five lines of text below the buttons are the

last five currently active messages with the latest message halt on the top. This part of the display is referred to as the top of the message stack or *call stack*. To continue, click *Debug* to open the Debugger.

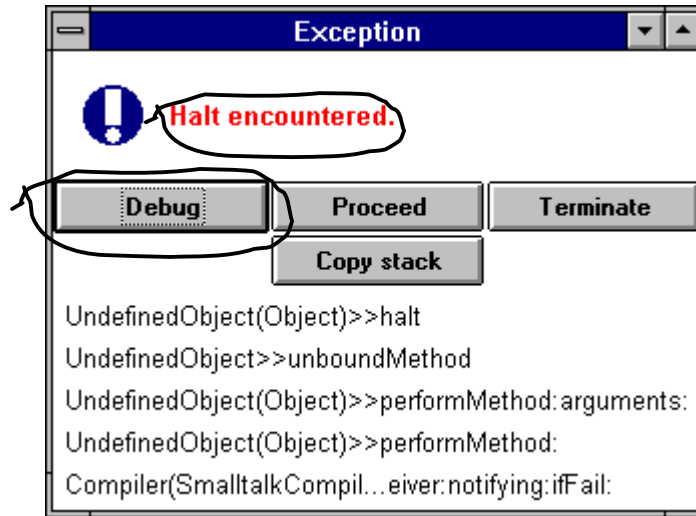


Figure 3.12. Exception window resulting from the execution of a halt message .

Clicking *Debug* opens the Debugger window in Figure 3.13. The scrollable top view of the window provides access to the call stack, the middle view is for displaying the code of a method selected from the stack, and the bottom part contains two inspectors. We will now explain these parts and illustrate their use.

As we already mentioned, the top message in the call stack is the method currently being executed, the one below is the method that sent the message on the first line, and so on. Each line consists of the name of the class of the object that received the message followed the name of the message. If the message is inherited, the class that defines it is given in parentheses. In our case, the currently active message is *halt*, it was sent by an *UndefinedObject*, but its definition is in the *Object* superclass of *UndefinedObject*.

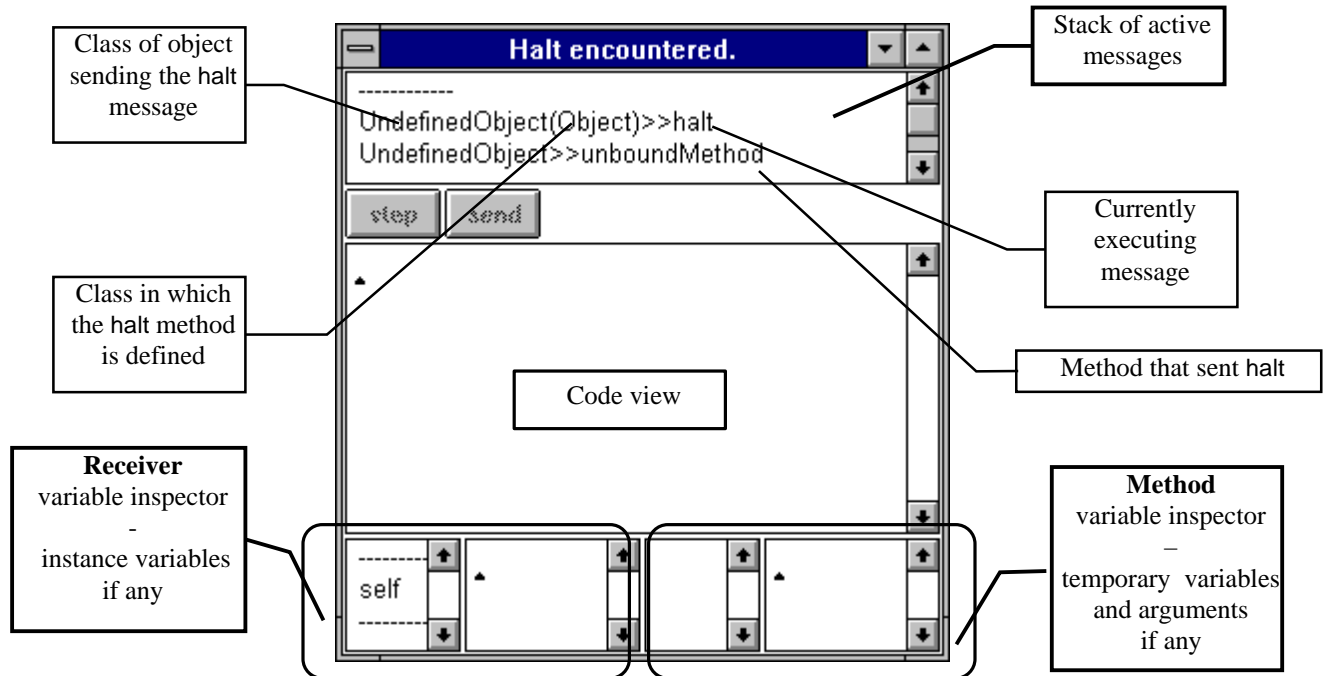


Figure 3.13. Debugger window with two most recently invoked methods shown on top.

To see the code of a message and the point reached in its execution, click the appropriate line in the call stack. Since a code fragment is always referred to as `unboundMethod`, we click the second line from the top and the Debugger changes to Figure 3.14. The message currently being executed is highlighted and the inspector at bottom right provides access to all temporary variables, arguments (none), and instance variables (none).

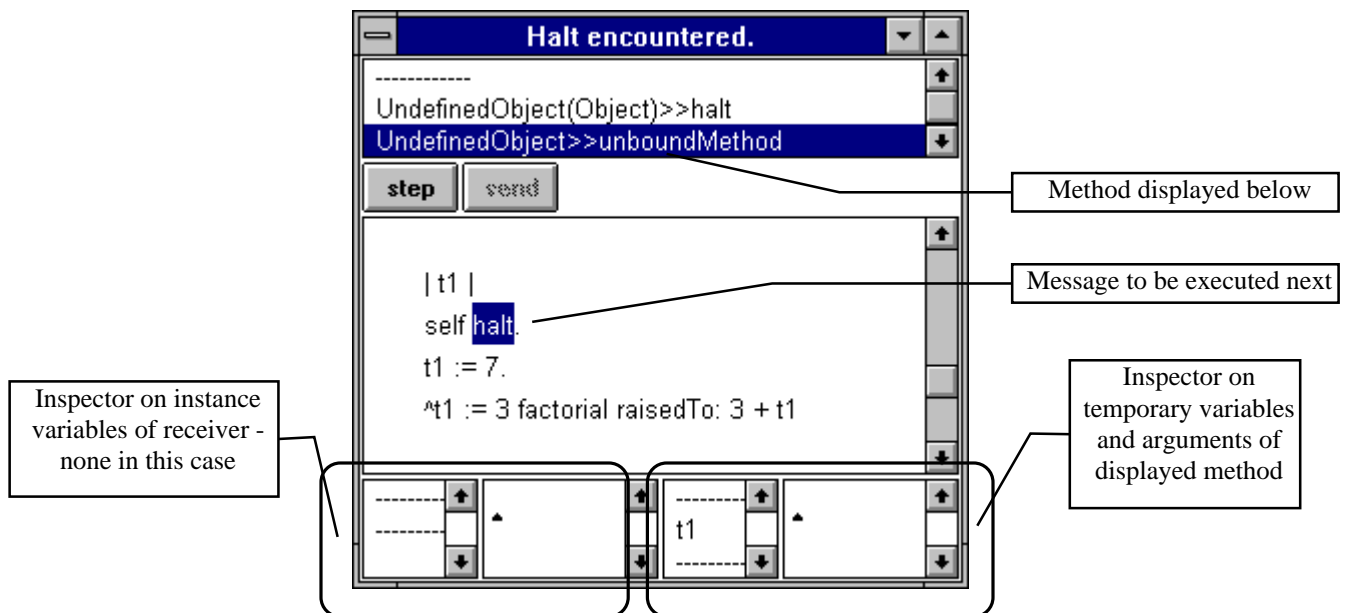


Figure 3.14. Debugger showing executing method and highlighting its current state of execution.

As you can see, the code is exactly like our code fragment except for variable names. This is because when the Debugger gets the message to display the code fragment, it reconstructs the code from a

condensed representation which does not include variable names, and it thus creates its own variable identifiers such as t1 and t2. The process of converting compressed representation back to the fully expanded *source code* is called *decompilation* because it is the reverse of compilation and the comment at the beginning of the code tells you that this is decompiled code. Decompilation always happens with code fragments so you can ignore the warning. It should not, however, happen with methods from the library; if it does, the directory path to your Smalltalk files are incorrectly set and must be changed on the *Sources* page of the *Settings* command under *Files* in the *Launcher* window.

Another difference between the original program and the code in the Debugger is the *circumflex* symbol ^ (called the *return* symbol in Smalltalk) in front of variable t1 on the last line. A caret in a method tells Smalltalk to stop executing the method at this point and return the object obtained by the expression following it, in this case the value of t1. In our example, the circumflex symbol was inserted by the compiler. We will ignore its use for now.

Back in the Debugger, we now have several options what to do next: we can continue execution within the Debugger, close the Debugger and continue execution of the program, or close the Debugger and terminate execution.

If we choose to *continue execution within the Debugger*, we can either click the *step* button or the *send* button: If we click *step*, the Debugger will execute the highlighted expression, update the inspectors, and proceed to the next message. If we click *send*, the Debugger will go inside the definition of the message being sent and execute it under your control. In other words, *send* allows you to get deeper and deeper into the execution of the code whereas *step* keeps you at the current level.

The remaining main options are *to close the Debugger window* (this will terminate the execution of the current message and the whole program that sent it), or to leave the Debugger and proceed with the execution of the program. To *proceed*, select *proceed* in the <operate> menu of the stack view.

In our case, we want to continue execution in the Debugger but we don't want to get any deeper at this point and we will thus click *step*. First, however, select variable t1 in the bottom right inspector so that you can observe how its value evolves as we execute the program. (t1 in the Debugger corresponds to x in our program). The initial value of t1 is nil because we have not assigned any value to it yet.

Click *step* a few times to execute a few steps until you reach the first assignment which assigns 7 to x and note the change in the inspector. The second assignment now begins execution by evaluating the expression on the right hand side. Smalltalk first checks for parenthesized expressions, does not find any, and looks for unary messages. It finds factorial, gets ready to execute it, and so on.

We leave the rest to you. Note, however, that the Smalltalk compiler sometimes generates code that does not evaluate in the order described by our four rules. This, however, does not change anything on the validity of our interpretation – although the code may be evaluated in a slightly different order, it will still produce the same result.

Other important properties of the Debugger and the Inspector

The code view in the middle of the Debugger can execute commands such as *do it*, *inspect*, and *print it* on any part of the code in the window. The Debugger also lets you change the code, recompile it with *accept*, and restart from the beginning of the method. This feature is frequently used because one often discovers mistakes while debugging, and on-the-spot modification and continued debugging allows you to test that a correction works. Note, however, that if the execution of the original code changed the values of some variables, execution will continue with these new values - unless you reset them using the inspectors at the bottom. To change the value of a variable or an argument, select it in the appropriate inspector, type the new value in the value part of the inspector window, and *accept* it from the <operate> menu. Note that you can also do this by evaluating an expression typed into the inspector view.

Main lessons learned:

- The Debugger has been designed for finding and correcting errors in code but it can also be used for step-by-step execution at various levels of detail.
- The Debugger can be closed at any point, and execution terminated or continued in normal way.

- The top window in the Debugger shows the call stack, a list of currently active messages with the one currently executing on the top.
- The code view of the Debugger has all code view properties found in the Workspace.
- Inspectors in the Debugger window show values of instance variables, method arguments, and temporary variables. All can be changed during debugging.
- The code displayed by the Debugger can be edited and recompiled, and its execution resumed from the start of the method.

Exercises

1. Modify the following code fragments to open the Debugger just before the last statement and trace its execution step-by-step. What is the result of the last statement?
 - a. `|x y|`
`x := 17.`
`y := (Dialog request: 'Enter a number' initialAnswer: '') asNumber.`
`(y+ 4) factorial / 2*x squared`
 - b. `|arg1 arg2|`
`arg1 := (Dialog request: 'Enter the sin argument initialAnswer: ') asNumber.`
`arg2 := (Dialog request: 'Enter the cos argument initialAnswer: ') asNumber.`
`arg1 sin between: 0.4 sin and: arg2 cos squared`
2. Repeat Exercise 1.a and when the Debugger window opens, modify the 'Enter a number' part of the third line of the code to read 'Enter the value of y'. *Accept* the new text and continue. Comment on the result.
3. Repeat Exercise 1.b and when the Debugger window opens, change the value of variables `arg1` and `arg2` using the Debugger's inspector and continue.
4. Repeat Exercise 1 and when the Debugger window opens, re-execute the last Dialog message within the Debugger by *inspect*. Comment on the result.
5. Execute expression `Date today` in the Debugger. Use the inspector to examine the instance variables of the class object `Date` (they are the *class* variables from the definition of class `Date`).
6. Repeat the previous exercise with expression `Time now`.

3.9 Cascading

In this section, we will explain cascading, a Smalltalk feature that saves typing when a sequence of messages is sent to the same receiver. We will start by demonstrating why it may be useful, using an example that will also show how to output intermediate results.

In the previous section, we used the Debugger to monitor the execution of code fragments by adding breakpoints. Another way to track program execution is to let the program write intermediate results to the Transcript window, the bottom area of the launcher. VisualWorks itself uses the Transcript to display information about some aspects of its internal operation and since the Transcript is easily accessible from any program, Smalltalk programmers often use it for debugging or for intermediate output, before creating a proper user interface for an application.

To write to the Transcript, send messages to the `Transcript` object. The most important messages that it understands are

<code>clear</code>	- to clear (erase) the Transcript view
<code>cr</code>	- to start a new line in the Transcript view (<code>cr</code> stands for carriage return)
<code>tab</code>	- to insert a tab in the text
<code>show: aString</code>	- to print <code>aString</code> in the Transcript view

As an example of the use of `show`:

Transcript show: 'This is a test of the show: message'

writes the string enclosed between apostrophes to the Transcript.

The `show:` message expects a string argument and if you want to print information about an object that is not a string - for example a number - you must first convert first it to a string. This is achieved by message `printString` which is understood by all Smalltalk objects because it is defined in class `Object`. As an example, to print number 37 in the Transcript, use

```
Transcript show: 37 printString
```

where `37 printString` converts number 37 to `'37'` which is a string. Message `printString` is also used by the `print it` command and by the inspector (to display the self object) and you are thus familiar with the kind of output that it produces.

As another example, we will now show the use of the Transcript for intermediate results. The following code fragment is a simple tax calculation that requests data from the user and records intermediate results in the Transcript:

```
| income expenses tax |
"Request income and expenses."
income := Dialog request: 'Enter income' initialAnswer: ".      "Reads a string."
income := income asNumber.      "Converts the string to a number."
expenses:= Dialog request: 'Enter expenses' initialAnswer: ".
expenses:= expenses asNumber.
"Calculate tax."
tax := (income - expenses) * 0.17.
"Display input and results in Transcript."
Transcript clear.
Transcript show: 'Income '.
Transcript show: income printString.
Transcript cr.
Transcript show: 'Expenses '.
Transcript show: expenses printString.
Transcript cr.
Transcript show: 'Tax '.
Transcript show: tax printString
```

The output produced in the Transcript window for income 20,000 and expenditure 4,700 is as follows:

```
Income 20000
Expenses 4700
Tax 2601.0
```

Enter the code fragment into a Workspace and execute it to see how it works. Note that the comma separating thousands should not be typed as input.

Save typing by cascading messages

The tax program that we have just written ends with a long sequence of messages to the Transcript. To reduce the clutter and to minimize typing, use *cascading*: When the same object is the receiver of a sequence of consecutive messages, you don't have to repeat it. Instead, replace the period at the end of the statement with a semicolon and delete the repeated receiver. With cascading, our original program can be written much more compactly as

```
| income expenses tax |
"Request income and expenses."
income := Dialog request: 'Enter income' initialAnswer: ".
income := income asNumber.
expenses:= Dialog request: 'Enter expenses' initialAnswer: ".
expenses:= expenses asNumber.
"Calculate tax."
tax := (income - expenses) * 0.17.
```

"Display input and results in Transcript."

```
Transcript clear;  
  show: 'Income ';  
  show: income printString;  
  cr;  
  show: Expenses ';  
  show: expenses printString;  
  cr;  
  show: 'Tax ';  
  show: tax printString
```

We used indentation to make it quite obvious that all the cascaded messages are sent to Transcript. Smalltalk programmers use cascading often because it saves typing and can make programs more readable. One place where cascading is used very often is in creating and initializing objects. As an example, a method creating a student object might contain the following cascaded statement:

```
self new firstName: firstName; lastName: lastName; id: aNumber
```

All the cascaded messages are sent to the object resulting from `self new` which shows that a sequence of cascaded messages can be sent to the result of an expression. Some programmers misunderstand how cascading works in this situation and either write programs that don't work correctly or don't use cascading. The precise rules for cascading are as follows:

- Cascaded expressions are separated by semicolons.
- All cascaded message have the same receiver as the last message *executed* before the first semicolon.

In our last example, the last message executed before the first semicolon is `firstName: firstName`, its receiver is `self new`, and all the remaining cascaded messages thus go to the result of `self new`. This calls for a look at another aspect of cascading: Although all the remaining messages go to the same object, the state of this object changes as the message executes. At the beginning, the new object is presumably uninitialized. After `firstName: firstName`, its `firstName` component is initialized to `firstName` and the following `lastName:` message thus goes to this modified object. After `lastName:` the value of `lastName` is changed and the `id:` message thus executes on the receiver in this again different state.

This may raise the question why we call all these receivers 'the same object'. The answer is that the receiver is the same in the same sense that I am the same person as I was yesterday (but one day older and surely different in some other ways as well), that a moving car is the same object as the same car when it was parked, and so on. The concept of object identity is important and we will return to it later.

Main lessons learned:

- Transcript is a part of Visual Launcher that can be used to output text. It is often used to output intermediate results during testing.
- The most important messages to the Transcript object are `clear`, `cr`, `show:`, and `tab`. The `show:` message requires a string argument.
- Any object can be converted to a string, by the `printString` message.
- Cascading is a shortcut technique allowing you to leave out the name of the receiver when a sequence of consecutive statements uses the same receiver.
- To cascade, include the receiver in the first statement but leave it out in the following statements and replace the periods separating the consecutive statements with semicolons.
- Cascading is often misunderstood and this may be why it is not used more frequently. The precise rules for cascading are as follows:
Cascaded expressions are separated by semicolons.
- All cascaded message have the same receiver as the last message *executed* before the first semicolon.
- Although all cascaded messages are sent to the same receiver, the state of the receiver may change as the cascade executes.

- Cascading without carefully checking who is the repeated receiver is a frequent cause of errors.

Exercises

1. Message `printString` works by passing `storeOn:` down the structure of composite objects. This illustrates the operation of messages and we suggest that you trace the execution of the following expressions and write a description of the execution of the `printString` message focussing on the passing of control to subordinate `storeOn:` receivers.
 - a. `self halt. (13@14) printString`
 - b. `self halt. 'abc' printString`
2. Predict and then confirm the result of `5 factorial factorial` and of `5 factorial; factorial`. Repeat for `5 + 5 * 5` and `5 + 5; * 5`.
3. In the following code fragment, add code necessary to clear the Transcript and display the value of `x` on a new line after each statement. Use cascading whenever possible. Execute the program with *do it*.

```
| x |  
x := 13.  
x := x + 3 * 5.  
x := x / 5.  
x := factorial
```
4. Repeat the previous exercise but type the code of the program into the Transcript view itself and execute it with *do it*. Then explain why it is better to use the Workspace to execute code fragments.
5. Write an expression solving the following problem and test it.
 - a. Ask the user to select a rectangle on the screen and print the coordinates of its upper left and lower right corners in the Transcript, one point per line with a proper legend. (Hint: Look at class `Rectangle`.)
 - b. Ask the user to enter the price of an item, calculate its provincial and federal tax according to Exercise 3.c of Section 3.5, calculate the final price, and print all input information and all results in the transcript.
6. Underline the receivers and give the results of the following cascaded messages. Explain.
 - a. `3 + 4 factorial; squared; sqrt`
 - b. `3 factorial + 4 factorial; squared; sqrt`
 - c. `3 between: 4 and: 5; + 7`
 - d. `3 factorial between: 4 and: 15; + 7`
 - e. `3 factorial between: 4 factorial and: 15 squared; + 7 squared`
 - f. `(Rectangle origin: 10@20 corner: 50@50) extent: 100@150; height: 40; width: 70; top: 40`
7. This example shows that although the identity of the receiver does not change as it is reused by consecutive cascaded messages, its state may change. Use the Debugger's inspector to find the consecutive states of the receiver after each message:
`'abc' at:1 put: $x; at: 2 put: $y; at:3 put: $z; printString`

3.10 Global variables, class instance variables, and pool dictionaries

In this section, we will introduce the remaining kinds of variables available in Smalltalk, starting with the following question what kind of object is Transcript. From its spelling, you would probably conclude that Transcript is the name of a class because it starts with a capital letter. In reality, Transcript is not the name of a class but the name of a *global variable*.

A global variable is a variable that is accessible from any Smalltalk code - any code fragment or any method in any class. This is quite unlike *temporary variables* which can be accessed only in the method in which they are declared, or *instance variables* which can be accessed only by instance methods of the class in which they are declared or its subclasses, or *class variables* which can be accessed only by instance and class methods in the class where the variable is declared or inherited. In other words, global variables have *global scope* whereas the scope of temporary, instance, and class variables is limited.

You are now probably wondering how Smalltalk knows about global variables if they are not defined in classes. The answer is that Smalltalk keeps the names of all globally accessible identifiers in a

special 'dictionary' object called *Smalltalk*⁷. This dictionary contains a list of 'keys', and for each of them its 'value' object. The keys are the globally shared identifiers, and the 'values' are their values. To see the contents of this dictionary, type and select the word *Smalltalk* and 'execute' it with *inspect*. You will get the inspector in Figure 3.15.

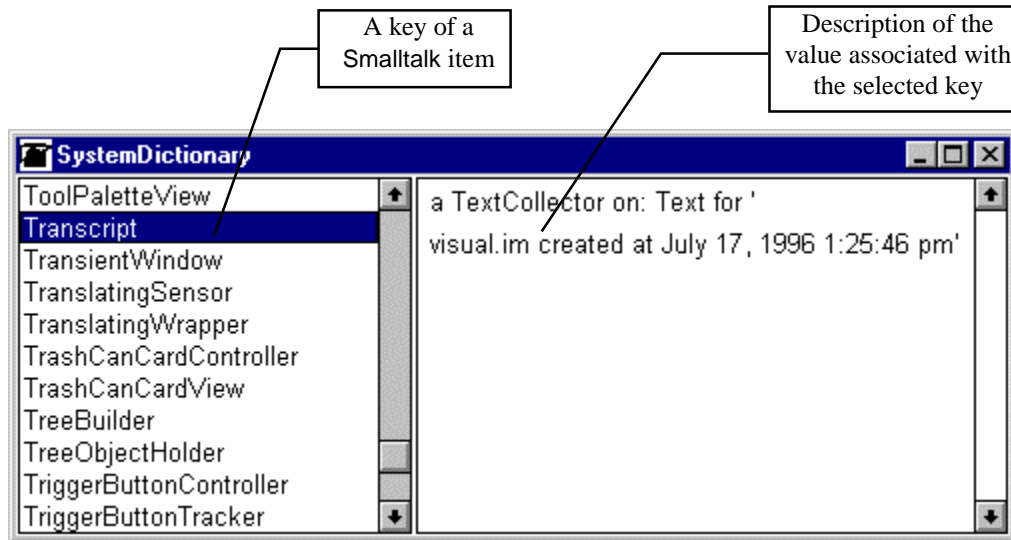


Figure 3.15. Inspector on the Smalltalk system dictionary obtained by executing *inspect* on *Smalltalk*.

The vast majority of keys in the Smalltalk dictionary are names of classes (in a sense, classes are also global variables because they are globally accessible) and their values are the class objects themselves. A few keys, such as *Transcript*, however, are global variables and their values are other kinds of objects. As an example, the value of key *Transcript* is an instance of class *TextCollector*, a class containing the definitions of *clear*, *show:* and all the other messages understood by *Transcript*. It is interesting to note that the identifier *Smalltalk* which refers to the Smalltalk dictionary is itself a global variable stored as a key in the Smalltalk dictionary.

How can you recognize a global variable? This is a tricky question but you could say that a global variable is a shared variable whose value is not a class. How can one create a global variable? One way is to type its name - for example *NewGlobal* - into some text space such as a *Workspace*, select it, and execute *do it* or *inspect*. This will open the notifier in Figure 3.16. If you select *global*, the identifier will be entered into the Smalltalk dictionary with *nil* as its value. You can then assign its value any time.



Figure 3.16. Creating a global variable via a notifier.

⁷ We will talk about dictionaries in Chapter 10.

You can also create a global variable with an initial value by evaluating an assignment such as

```
NewGlobal := 10
```

which will open a Notifier as in Figure 3.16 and create and initialize the variable upon confirmation.

Another way to create a global variable is to add it via the <operate> menu of the inspector of the Smalltalk dictionary; You can then enter its value in the right half of the inspector window and *accept*. To remove a global variable, use *remove* in the <operate> the menu of the Smalltalk inspector.

When are global variables used? The answer is - very rarely. The advantage of global variables is that they persist from one saved Smalltalk session to another and are restored when you open Smalltalk again. (This explains why you get your Transcript and your classes with their class variables back when you open a new session.) Because of their persistence, global variables can be used to hold objects that should last beyond the execution of a single application or code fragment. The Transcript object is a perfect example because it maintains a text view with a record of system activities that you may want to read when you reopen Smalltalk.

However, global variables also have disadvantages: For one thing, since persistent objects are carried along throughout the whole session and for all remaining sessions until you remove them, they may occupy a lot of memory. And if you forget about them, this memory is wasted. Another disadvantage of global variables is that they violate one of the basic principles of Smalltalk - encapsulation. According to the principle of encapsulation, the scope of accessibility of objects should be minimal. Global variables are accessible globally and control over their values is available all over the whole system. This makes them vulnerable and unsafe because their values can be changed uncontrollably by any method or code fragment. These two disadvantages of global variables are a sufficient reason why their use is strongly discouraged and why the Smalltalk system itself defines only a few of them.

In addition to its fundamental role of holding all existing classes, the Smalltalk object is very important because it is the root from which all checks for live objects begin: All objects directly or indirectly referred to by Smalltalk are live, all other objects can be garbage collected. More formally, we can define *active objects* as follows:

1. All objects accessed from the Smalltalk dictionary are active.
2. All objects referenced by active objects are active.

As a consequence of this definition, all class objects are active (and will not be 'garbage collected'). All objects referenced by classes (such as class comments, class variables, and class definitions) are also active because they are referenced by classes. All pool dictionary objects are also active. And all objects in pool dictionaries are active because they are referenced by pool dictionaries. All objects referenced by global variables are active because global variables are in Smalltalk. (This includes, among other things, all windows on the screen.) And all objects referenced by global objects are active, and all objects referenced by them are also active, and so on. If you can 'find a path' from the Smalltalk dictionary to an object, that object is active - and all other objects are inactive and will be garbage collected when the garbage collector is activated. Thousands of objects are normally active at any time.

By the way, the definition of an active object is said to be *recursive* because its second part defines what the concept *active object* means in terms of the concept *active object* itself. You could say that a recursive definition 'feeds on itself'. Recursive definitions are very important and we will encounter them again later.

Pool dictionaries

In addition to global variables and classes, the Smalltalk dictionary contains a few specimens of another kind of shared object - a *pool dictionary*. A pool dictionary is a dictionary (with keys and values - like Smalltalk) whose *keys* are directly accessible to a selected group of classes - those classes in which it is defined as a pool dictionary. Any method in each class in this group can directly access each pool dictionary key as if it were a class variable, just like a class or instance variable. Other classes can also access pool variables (because pool names are keys in Smalltalk) but not directly - their access requires going through

the pool dictionary's name. To create a pool dictionary, add its name to the Smalltalk dictionary as for a global variable, enter values into it (we will see how in the chapter on dictionaries), and enter the name of the pool dictionary as the pool dictionary keyword in each class that should have access to it.

As an example of a pool dictionary, TextConstants contains information needed for display of text. One of its keys is, for example, called Tab and its value is the code of the character that produces a tab in text. All classes that define TextConstants as their pool dictionary can access its Tab directly. As an example, TextConstants is a pool dictionary in class Text (Figure 3.17) and you can thus execute messages such as

char = Tab "Check whether variable char contains the Tab character."

in any method defined in class Text or its subclasses. Defining TextConstants as a pool dictionary eliminates the need to duplicate the information in each class that needs it, which would otherwise be necessary because these classes are not on the same branch of the hierarchy tree and cannot define these values as shared class constants. TextConstants provides a typical use of pool dictionaries – as a depository of useful constants whose access is faster than accessing them via methods.

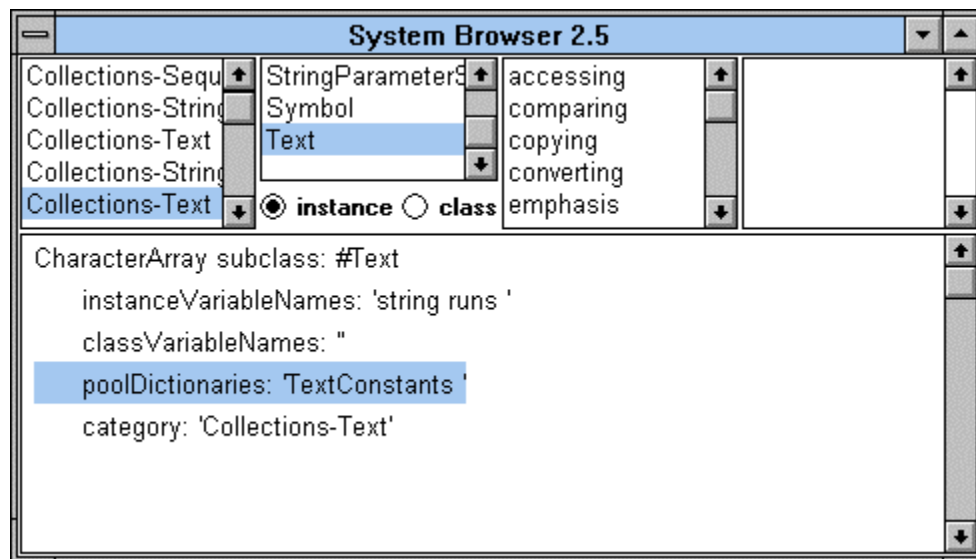


Figure 3.17. Class Text has a pool dictionary variable TextConstants.

To find where pool dictionary is used, open the browser on one of the classes that defines it, select the name of the pool dictionary in the definition, and execute *explain* in the <operate> menu. To find where a particular key (such as Tab) of a pool dictionary variable is used, open an inspector on the pool dictionary, find the pool dictionary variable key, and execute *references* from the <operate> menu.

Pool dictionaries have similar disadvantages as global variables, are rarely used, and their use is discouraged.

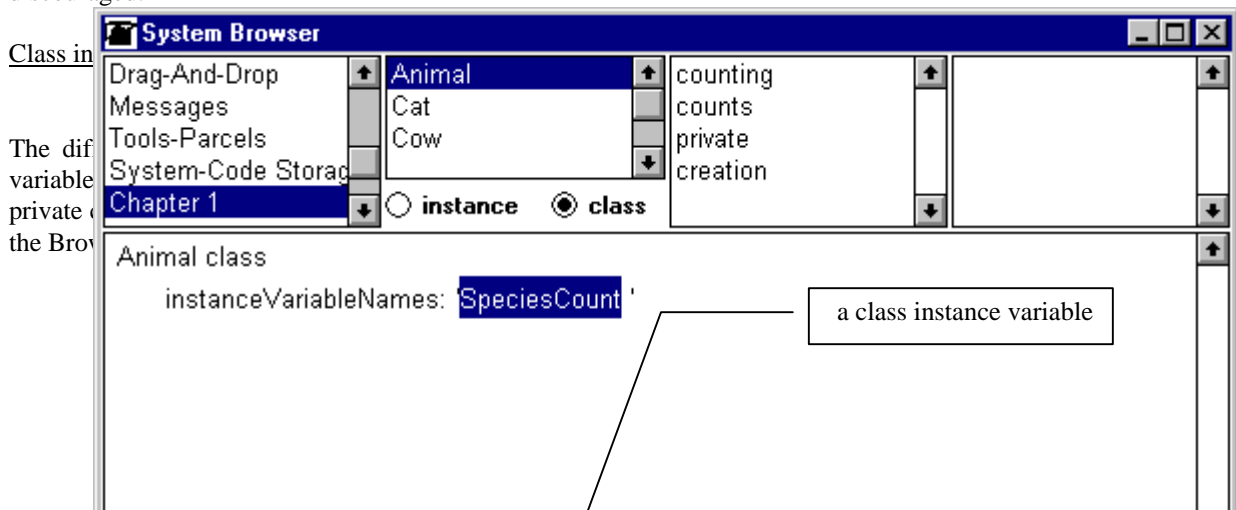


Figure 3.18. Class instance variables are listed on the class side of the browser.

Although class variables are rare, they are useful and *Farm 6* gives an example that illustrates their utility and the difference between class variables and class instance variables (Figure 3.19). The problem addressed by *Farm 6* is counting how many cats, how many cows, and how many dogs a farm has, and how many animals there are on the farm altogether. To do these calculations, we defined two variables in class *Animal*. One is a class variable called *TotalCount* and you can find it on the instance side of the definition of *Animal*. The other is a class instance variable called *SpeciesCount* and you will find it on the class side of the definition of *Animal* in the Browser as shown in Figure 3.18.

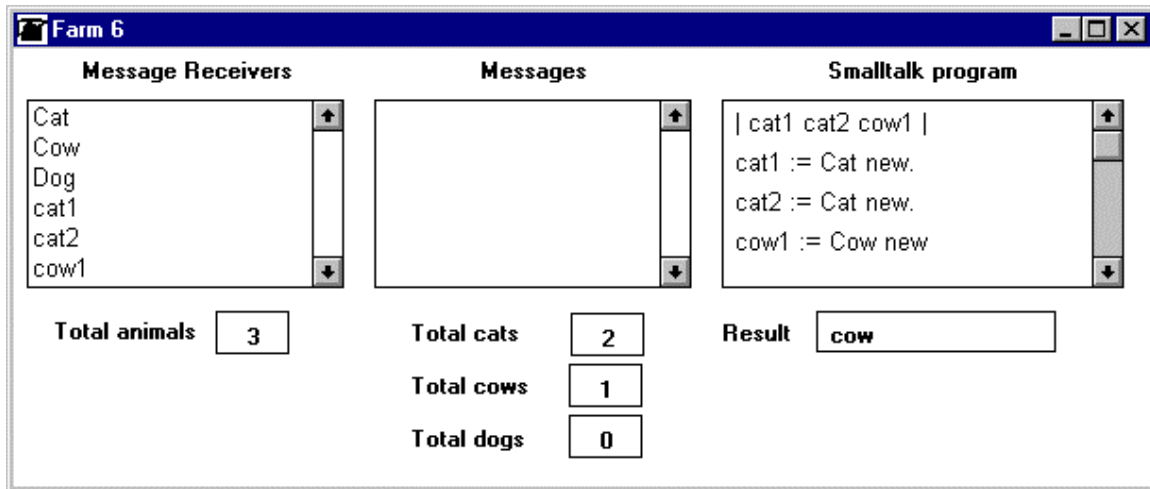


Figure 3.18. Farm 6 uses class variables and class instance variables for counting.

Both variables are inherited by *Cat*, *Cow*, and *Dog* and we increment them both when *Animal* gets the new message. Since all animals share the same copy of the class variable *TotalCount*, *TotalCount* is incremented every time that a new animal is created, whether it's a *Cat*, a *Cow*, or a *Dog*. As for class *instance* variable *SpeciesCount*, each animal class inherits its own *private* copy which means that *Cat*'s *SpeciesCount* is incremented only when a new *Cat* is created, a *Cow*'s *SpeciesCount* is incremented only when a new *Cow* is created, and a *Dog*'s *SpeciesCount* is incremented only when a new *Dog* is created. In this way, *TotalCount* keeps count all animals whereas *SpeciesCount* keeps count for each animal separately. This is a typical use for class variables and class instance variables.

Main lessons learned:

- A global variable is a variable accessible by any Smalltalk code and its description is stored in Smalltalk, the System Dictionary. Smalltalk is a global variable itself.
- Smalltalk elements are key-value pairs and all their keys are globally accessible across all Smalltalk code.
- Shared variables include names of classes, pool dictionaries, and global variables. Their identifiers start with a capital letter.
- The Smalltalk object is persistent, surviving beyond the execution of a program fragment or a Smalltalk session. Its contents are saved in the image file whenever you save the Smalltalk environment, and restored when you start a new session. All variables stored in it and their values are thus also persistent.
- Class instance variables are similar to class variables but their value is private to a class. This is different from class variables whose values are shared by the class and all its subclasses.

- A pool dictionary is a dictionary of variable-value pairs. Its variables can be directly accessed by all classes that define the dictionary as their pool dictionary. Its typical use is as a depository of useful constants shared by several classes unrelated in the class hierarchy.
- Although global variables and pool dictionaries have their advantages, their use beyond a few well-justified situations is discouraged because they violate encapsulation and require continuous allocation of significant memory space.
- An object is active if a link to it can be traced from the Smalltalk dictionary.
- Active objects are those that have a connection to Smalltalk. All other objects are inactive and will be garbage collected.
- A definition is recursive if it defines a concept in terms of itself.

Exercises

1. Inspect Smalltalk to find all references to Transcript.
2. Inspect the global variable Smalltalk in the Smalltalk inspector.
3. Inspect TextConstants with the inspector and list five interesting components.
4. Find all classes that define TextConstants as their pool dictionary variable. Find all references to key Tab in the TextConstants pool dictionary variable.
5. Examine 20 randomly selected keys in Smalltalk and find their values. How common are pool variables and global variables compared to classes?
6. Add global variable NewGlobal with value 7 to the Smalltalk and check that you can access it from any Workspace and from the Transcript. Change its value to 35. Save the image and exit Smalltalk, reopen Smalltalk and check that NewGlobal is still there with the same value, delete it, exit with saving, restart, and check that NewGlobal is gone.
7. Are any other kinds of shared objects stored in the Smalltalk dictionary?
8. Use *Farm 6* to find out about class and class instance variables.
9. Enact the *Farm 6* program shown above, assigning the role of Animal, Cat, Cow, and Dog classes to individual students and focusing on the behavior of TotalCount and SpeciesCount.
10. Examine 30 randomly selected classes and count how many of them have a class instance variable.

Conclusion

A Smalltalk code fragment consists of one or more statements separated by periods. Statements may be message send or assignment expressions.

Names of variables and methods are called identifiers. They are formed according to the following rules:

- An identifier must begin with a letter and the remaining characters must be letters or digits.
- In principle, identifiers may contain any number of characters. In reality, there is an implementation limit on the length but this limit is very high.
- Identifiers are case sensitive.
- Names of methods and temporary and instance variables start with a lowercase letter, names of classes and other shared variables start with an uppercase letter.
- The following words are reserved for special uses and cannot be used for identifiers: self, super, true, false, nil, and thisContext.

Smalltalk programmers choose identifiers very carefully to make programs as readable as possible. For additional documentation, use comments. Any text surrounded by double quotes is a Smalltalk comment and is ignored when the program is executed.

Smalltalk recognizes three types of messages - unary, binary, and keyword. Their names are formed according to the following rules:

- A unary message consists of an identifier and does not have any arguments.
- A binary message consists of one or two special symbols selected from the + - / \ * & @ , > < = and other special characters and has one argument. Symbol – combination – is not allowed.
- The selector of a keyword message consists of one or more keywords and each keyword is an identifier followed by a colon. In a message, each keyword must be followed by an argument.
- All unary messages and all keywords begin with a lowercase letter.

Selector refers is the full name of a message without arguments. Selectors are used in browser searches and in certain programming contexts.

To make it possible to refer to an object in several places in a code fragment or a method, assign it to a temporary variable using an assignment. Don't assign an object to a variable unless you need to refer to

it later in the program or unless the variable will make the code more readable. All variables must be declared before the first statement.

The scope of a temporary variable (the range in which it can be used) is the code fragment or method in which it is declared. An object bound to a temporary variable will be destroyed when the execution of the code in which the variable is defined is finished, unless it is bound to another object which survives.

In addition to temporaries, Smalltalk defines several other kinds of variables. Instance variables capture object state and each instance of a class has its own private copy; instance variables are inaccessible to other objects. Class variables are shared by the class in which they are declared, its instances, its subclasses, and their instances.

The lifetime of objects referenced by instance and class variables is the same as the lifetime of their parent objects. Since classes are persistent objects (survive from one saved Smalltalk session to another), objects bound to class variables are also persistent.

Code fragments are used to test pieces of code before they are used in methods. Although code fragments can be executed in any text view, Smalltalk programmers normally use a Workspace window for this purpose. To execute a code fragment, select it using the <select> button, and execute it using *print it*, *inspect*, or *do it* via the <operate> button. For execution, Smalltalk converts a code fragment to the equivalent of a temporary method and in this sense, a code fragment is thus a method.

Understanding the rules of formation and evaluation of Smalltalk expressions is essential for reading and writing Smalltalk code. The rules are simple: Everything is evaluated from left to right, parenthesized expressions first, unary messages next, binary messages next, and keyword messages last.

When a series of consecutive expressions are sent to the same receiver, Smalltalk programmers use cascading. Cascaded expressions are separated by semicolons and the receiver is not repeated. All expressions are executed by the receiver of the last message executed before the first semicolon.

Global variables are globally accessible, which means that they can be accessed from any method or code fragment. Pool dictionaries can be global in the same sense but their components can be accessed directly from classes in which they are named as pool variables. Class instance variables are similar to class variables but they are not shared by the defining class and its subclasses; instead, each of them has its own copy. Global variables, pool variables, class variables, and class instance variables are classified as shared variables which means that they can be accessed by more than a single class or its instance. Names of all shared variables begin with a capital letter and their values are stored in the Smalltalk System Dictionary. They are all persistent because they are kept in Smalltalk during the whole session, saved by *save as* and *save and exit*, and restored when a new session is opened.

All objects that can be traced from the Smalltalk dictionary are active. All other objects are inactive and will be collected by the automatic garbage collector.

Terms introduced in this chapter

assignment - consists of variable name, assignment symbol, and an expression; assigns binding to result of expression to the variable; returns the value of expression

binary message - message whose selector consists of one or two special symbols; has one argument

binding - link between a variable and the object attached to it

blue button - old name for <window> button

call stack - list of currently active messages in the order in which they have been activated

cascaded message - a statement consisting of a receiver and expressions separated by semicolons; all expressions are sent to the receiver of the last message executed before the first semicolon

case sensitivity - distinguishing the difference between upper and lower case characters in identifiers; Smalltalk identifiers are case sensitive

class instance variable - similar to class variable but each subclass has its own private copy

class variable - variable associated with a class and accessible to all its instance, class, and subclass methods; a single copy of the corresponding object is shared by all objects that can access it

code fragment - one or more statements constructed for testing purposes; an intermediate step in developing Smalltalk applications; in execution, treated as a method

compilation - the process of converting Smalltalk code into internal computer representation for execution; automatically performed when a code fragment is executed or method definition *accepted*

comment - explanatory note inserted into a program for documentation, has no effect on program execution; in Smalltalk, any text surrounded by double quotes

Debugger - tool used to analyze malfunctioning programs or to gain understanding of the behavior of code execution; provides step-by-step evaluation and access to variables

decompilation - the reverse of compilation

do it - <operate> menu command; executes the selected (highlighted) code

global variable - variable accessible from any method; stored with its value in the System Dictionary

identifier - name of variable, message, or keyword; Smalltalk identifiers are case sensitive, must begin with a letter, and contain only letters or digits; the length of a Smalltalk identifier is unlimited

inspect - <operate> menu command; executes selected code and opens Inspector on the result

Inspector - window for viewing and editing the components of an object

instance variable - internal variable of an instance of a class; a separate copy is owned by each instance and is not directly accessible to other objects

keyword - identifier followed by a semicolon

keyword message - message whose name consists of a sequence of keywords followed by arguments

nil - special object bound to variables before they are assigned a value by an assignment; instance of class UndefinedObject

object lifetime - span of time during which an object is active and accessible to other objects

persistence - object's ability to survive from one session to another; Smalltalk objects are not persistent unless they are stored in Smalltalk or directly traceable to it

pool dictionary - a globally accessible dictionary of variable names and values; variables are accessible directly in classes that specify the dictionary as their pool dictionary

print it - <operate> menu command; executes the selected code and prints a description of the result

recursive definition - definition defining a concept in terms of itself

red button - old name for <select> button

rules of evaluation - rules used to determine the order of execution of components of a combined message

scope - range of code in which an identifier is recognized

selector - the name of a message as used in message searches; does not include arguments

shared variable - variable accessible to more than a single object; includes global variables, class variables, class instance variables, pool dictionary variables, and pool dictionary variables

Smalltalk - the single instance of SystemDictionary holding the names and values of all shared variables; saved with every save operation and restored when a Smalltalk session is opened

statement - expression followed by a period, or the last expression in method definition

temporary variable - variable defined at the beginning of a method

Transcript - text view attached to the bottom of the Visual Launcher; used by the system to output certain messages and by programmers to output intermediate results during testing

unary message - message consisting of an identifier; does not have any arguments

variable - identifier used to refer to an object; initially automatically bound to *nil*

variable declaration - list of all variables used in a method; must precede the first statement

<window> button - the rightmost button of the mouse

<window> menu - pop up menu of the <window> button; contains window-related commands

Workspace - window used by Smalltalk programmers to execute code fragments

yellow button - old name for <operate> button

Chapter 4 - True and False objects, blocks, selection and iteration

Overview

In this chapter, we begin exploring the most important classes of Smalltalk. We start with classes representing true and false objects because they are needed to make choices and to control repetition, and this is required in almost all programs.

The use of true and false objects heavily depends on the concept of a block. In essence, a block is a sequence of one or more statements that are always executed together. In the case of program choices and iteration, each alternative action or sequence of iterated statements is represented by a block which is executed or skipped depending on the result of evaluating a condition with a true or false result.

4.1. Why we need true and false objects

Most problems include questions that must be answered to determine what to do next. As an example, the response to a request for a book from a library catalog depends on the answer to questions such as 'Is the book in the library?' and 'Is the borrower allowed to take the book out?'. As another example, a program printing paychecks processes a list of employees, constantly asking the question 'Is there another employee record to print?' When the answer is 'no', processing stops.

In English, answers to questions such as these are usually in the form of 'yes' and 'no' but programming languages use the terms 'true' and 'false'. The two forms are equivalent because a question such as 'Is the book in the library?' can be formulated as 'Is it true that the book in the library?' and 'yes' and 'no' answers then naturally translate to 'true' and 'false'.

Smalltalk 'true' and 'false' objects are implemented by classes True and False. These two classes share many properties and they are thus defined as subclasses of the abstract class Boolean (Figure 4.1). The name Boolean or 'logic' is also used when we talk about the true and false objects in general¹.

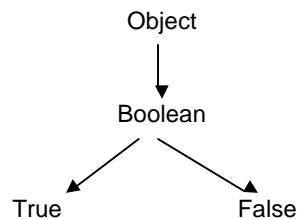


Figure 4.1. Booleans in Smalltalk class hierarchy.

As an illustration of the use of Booleans, consider a program that calculates tax. Assume that earners of incomes up to \$7,000 pay no income tax while higher income earners pay 7% of their income above \$7,000. The tax calculation program will read the income and ask: 'Is it true that the income is not above \$7,000?' If the answer is 'true' the program will take one course of action (tax = 0) but if the answer is 'false' the program will calculate a tax. This could be implemented as follows:

1. Set tax to 0.
2. Is income is greater than \$7,000? If the answer is 'true', change tax to 7% of income over \$7,000.

In a more realistic tax program there would probably be more tax brackets such as no tax up to \$7,000, 7% on amounts between \$7,000 and \$15,000, 12% on amounts between \$15,000 and \$30,000, and 25% on amounts exceeding \$30,000. The logic of this calculation is more complex but still based on true/false questions:

¹The name Boolean is used to honor the 19th century British mathematician George Boole who developed mathematical foundations of logic.

1. Initialize tax to 0.
2. Is income between 0 and 7,000? If the answer is 'true', calculation is finished.
3. Calculate 7% of income between 7,000 and 15,000 and add this amount to tax.
4. Is income less or equal to 15,000? If the answer is 'true', calculation is finished.
5. Calculate 12% of income between 15,000 and 30,000 and add this amount to tax.
6. Is income less or equal to 30,000? If the answer is 'true', calculation is finished.
7. Calculate 25% of income exceeding 30,000 and add this amount to tax.

As another example of the use of Booleans, a program that copies files from one disk to another might work on the following principle:

1. Is there is another file to be copied?
2. If the answer is 'false', execution is finished.
3. If the answer is 'true', copy the file and return to Step 1 to repeat the sequence.

These examples show that Boolean objects are needed to

- decide whether to take an action or not,
- decide which of several possible alternative courses of action to take, and
- decide whether to repeat a sequence of actions.

Class Boolean and its subclasses True and False are essential in all these operations and implement some of the corresponding methods. The remaining ones are left to a few other classes that will be introduced later.

Input of Boolean values

Before we introduce the messages that operate on Boolean objects, we will show how a user can input a Boolean value by using a *confirmer window*. To produce this window, class Dialog contains the class message confirm: which returns an instance of True (the true object) or an instance of False (the false object) depending on the user's choice. As an example,

Dialog confirm: 'Do you want to close the file?'

opens the window in Figure 4.2 and returns true if the user clicks *yes* or hits the <Enter> key; it returns false if the user clicks *no*.

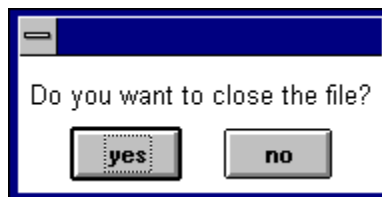


Figure 4.2. The confirm: message opens a confirmer window with a message-specified prompt.

Main lessons learned:

- Classes Boolean, True, and False are the basis for deciding whether an action should be taken or not, which of several actions should be executed, and whether an action should be repeated or not.
- Class Dialog provides message confirm: for input of Boolean values. It opens a confirmer window which solicits a true/false response in the form of a *yes* or *no* answer.

Exercises

1. Formulate an algorithm for finding the smallest of three numbers.
2. One way to calculate an integer approximation k of the square root of a positive integer n is as follows: Start with $k = 1$, and keep incrementing k by 1 until its square exceeds n . Subtract 1 from k to get the approximation. Formulate this principle as an algorithm.
3. Describe a situation where Booleans are needed to decide
 - a. whether to execute an action
 - b. which of two alternative actions to take
 - c. whether to repeat a sequence of actions one more time
4. Write an expression to open a confirmer with the text 'Do you understand Booleans?' and execute it with *inspect* and *print it* to see the two possible results.
5. In addition to `confirm:`, Dialog contains a related message called `confirm:initialAnswer:`. Look it up in the System Browser and write an example expression to show how it works.

4.2 Boolean messages for deciding whether to take an action or not

The simplest use of Booleans is for deciding whether to take a specified action or not. Smalltalk contains two messages called `ifTrue:` and `ifFalse:` for dealing with this situation and calls them *controlling* messages. As an example,

(Dialog confirm: 'Do you want to see 500 factorial?')	"Returns true or false."
ifTrue: [Transcript clear; show: (500 factorial printString)]	"Executed for true receiver."

asks the user for a true or false answer and executes the bracketed statements if the result is true. The bracketed statements are ignored if the confirmer returns false. Message `ifFalse:` has the opposite effect. More formally, the two messages have the form

aBoolean ifTrue: aBlock
aBoolean ifFalse: aBlock

where `aBlock` is an instance of class `BlockClosure`, a sequence of statements surrounded by square brackets such as

[Transcript show: 'Test'; cr]

Evaluating a block returns executes all statements in the block and returns the result of the last statement executed in the block. (A more accurate description will be given later.)

Execution of `ifTrue:` can be described diagrammatically by the *flowchart* in Figure 4.3 and `ifFalse:` follows the obvious alternative pattern.

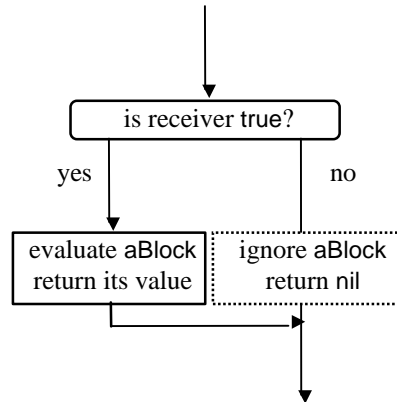


Figure 4.3. Evaluation of ifTrue: aBlock.

We will now illustrate ifTrue: and ifFalse: on several examples.

Example 1: Giving the user a choice

Problem: Write a code fragment to ask the user whether to delete a file or not. If the answer is 'yes', the open the *notifier window* in Figure 4.4.



Figure 4.4. Notifier window of Example 1.

Solution: To display a notifier window, send warn: to class Dialog as in

Dialog warn: 'You will not be able to recover the file later!'

The complete program is as follows:

```
| answer |  
"Ask the yes/no question and store the answer in variable answer."  
answer := Dialog confirm: 'Do you want to delete the file?'.  
"If the answer is yes, display notifier window."  
answer ifTrue: [Dialog warn: 'You will not be able to recover the file later!']
```

Type the program into a Workspace and execute it. Note that you don't have to type the ifTrue: keyword, just press <Ctrl> t (for 'true') and Smalltalk will insert the ifTrue: text; similarly <Ctrl> f produces ifFalse:. Note also that we could rewrite the code without the variable as follows:

```
"Ask the yes/no question and if the answer is yes, display notifier window."  
(Dialog confirm: 'Do you want to delete the file?')  
  ifTrue: [Dialog warn: 'You will not be able to recover the file later!']
```

This form is quite readable and should not cause any misunderstandings; we thus prefer it.

Example 2: Converting text to uppercase letters

Problem: Ask the user to enter some text (a string) and convert it to upper case if the user so desires. Print the resulting text in the Transcript window.

Solution: A message to display a window and ask the user for a string is available in class `Dialog` and its most common form is `request:initialAnswer:`. As an example

`Dialog request: 'What is your favorite beverage?' initialAnswer: 'Skim milk'`

produces the window in Figure 4.5. If the user clicks *OK* or presses <Enter>, it returns the displayed initial answer *'Skim milk'*. If the user types another answer and clicks *OK* or presses <Enter>, it returns the new string. If the user clicks *Cancel*, it returns an empty string with no characters.



Figure 4.5. Dialog request: 'What is your favorite beverage?' initialAnswer: 'Skim milk'

To write the program, we need to know how to convert a string to upper case. if we don't, we open class `String` (that seems the logical place) and search for a suitable method by executing command *find method* in the <operate> menu in its instance protocol sview. Unfortunately, the list that opens does not offer any suitable method. We thus try `String` superclass `CharacterArray` and find that its method list (Figure 4.6) includes method `asUppercase`. When we open its definition, we find that it indeed solves our problem. (Messages converting one type of object to a related form of object are among the most frequently used Smalltalk messages. They usually begin with *as*, as in `asUppercase` or `asNumber`.)

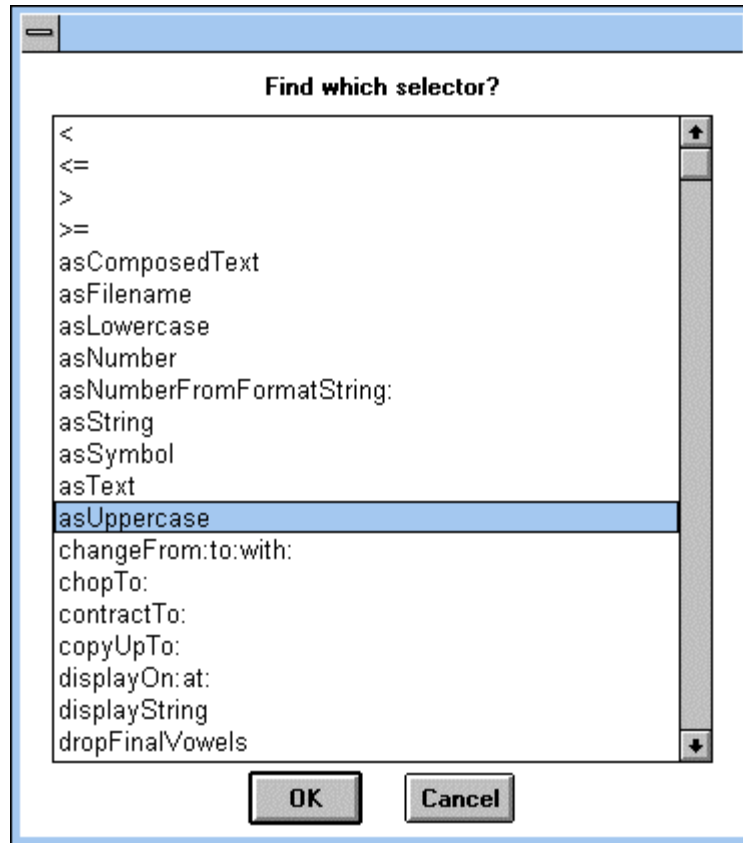


Figure 4.6. Response to *find method* in the instance protocol view of class *CharacterArray*..

We now have all necessary pieces to solve our problem and the complete solution is as follows:

```
| string convert |  
"Get a string from the user."  
string := Dialog request: 'Enter a string' initialAnswer: ''. "Display prompt without any initial string."  
"Ask whether user whether to convert the string to upper case."  
convert := Dialog confirm: 'Do you wish to convert the text to upper case?'.  
"If the answer is yes, change string to upper case."  
convert ifTrue: [string := string asUppercase].  
Transcript clear; show: string
```

Type the program into a workspace and execute it trying both 'yes' and 'no' answers to test how it works.

Main lessons learned:

- To control whether to execute a block of statements or not, use `ifTrue:` or `ifFalse:` from classes `True` and `False`. Both use a block as an argument.
- A block is a sequence of zero or more statements surrounded by square brackets. A block is an object, an instance of class `BlockClosure`. Evaluating a block evaluates the statements in the block and returns the result of the last statement.
- When the receiver of `ifTrue:` is a true object, the block argument is evaluated and the result returned. When the receiver is false, the block is ignored and `nil` is returned. Message `ifFalse:` has the opposite behavior.
- Messages converting one type of object to another are among the most frequently used Smalltalk messages. They usually begin with `as`, as in `asUppercase` or `asNumber`.

Exercises

1. Write a code fragment asking the user to choose whether to end program execution or not. If the answer is 'yes', nothing happens; if the answer is 'no', display a notifier saying 'Closing application.'
2. Every statement using `ifFalse:` can be converted to `ifTrue:` and vice versa by inverting the condition that calculates the receiver. Rewrite the following statements with the alternative message:
 - a. `x > 3 ifTrue: [Transcript clear]` "Rewrite using `ifFalse:`"
 - b. `x ~= y ifFalse: [Dialog warn: 'Wrong value of x']` "`~=` means 'not equal', 'equal' is `=`"
3. Check that when you replace the line
`convert ifTrue: [text := text asUppercase]`
in Example 2 with
`convert ifTrue: [text asUppercase]`
the program will not work. The reason is that expression `text asUppercase` returns a copy of the text object converted to uppercase *but does not change* the value of the receiver `text` itself. This is a common property of conversion messages that is frequently forgotten.
4. Count all conversion messages starting with `as` in the library that have the pattern `as...` message. Use *implementors of...* in the *Browse* command of the launcher with pattern `as*`. The `*` symbol is called a *wildcard character* and the search will use it to match any string starting with `as` such as `asUppercase`.
5. Some conversion messages don't have the form `as*` and we have already encountered one of them. Which one is it? (Hint: The message does not convert the receiver into a related object but produces an object that describes the receiver in a readable form.)

4.3 The definition of `ifTrue:`

The best way to understand how a message works is to study its definition. Let's examine method `ifTrue:` to see how it works. In class `Boolean`, the definition of method `ifTrue:` is as follows:

`ifTrue: alternativeBlock`

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is `nil`. Otherwise answer the result of evaluating the argument, `alternativeBlock`. This method is typically not invoked because `ifTrue:/ifFalse:` expressions are compiled in-line for literal blocks."

`^self subclassResponsibility`

Since this is our first method, let us first explain its structure. A method definition starts with a *heading* which defines the selector and names the arguments, if any. In this case, the heading is

`ifTrue: alternativeBlock`

the selector is `ifTrue:` and the argument is called `alternativeBlock`. The name `alternativeBlock` indicates that the argument should be a block, and that its execution is an alternative, a possibility.

Following the heading is usually a *comment* describing the purpose of the method and possibly how it works. We will have more to say about the comment in this definition in a moment. The comment may be followed by the definition of temporary variables for the method (none in this case), and by the *body* of the method - the part specifying how the method works. The body consists of a sequence of zero or more statements. A more formal definition of the rules of Smalltalk (its syntax) is given in Appendix 8.

Every Smalltalk method return an object and its default value (the value returned unless the program specifies otherwise) is the receiver. As an example, if the receiver of the message is a true object, the method returns `true` by default. If we want to return something else, we must calculate the object and put the caret symbol (the return operator `^`) in front of the expression that calculates the result - as in the definition of `ifTrue:`. The return operator also forces the execution of the method to stop at this point. This makes it possible to exit from any block anywhere inside a method, not just at the end.

After these general notes, let us now examine the body of the definition. The line

`^self subclassResponsibility`

says that the receiver of the message sends message `subclassResponsibility` to itself (object `self`) and returns the object returned by that method.

Message `subclassResponsibility` is defined in class `Object` and it is thus understood by all objects. It opens the Exception Window in Figure 4.7 telling you, in essence, that you cannot send this message to this object and that this method should be redefined in a lower level class. Message `subclassResponsibility` is frequently used in abstract classes (such as `Boolean`) when they leave the implementation of a shared message to concrete subclasses.

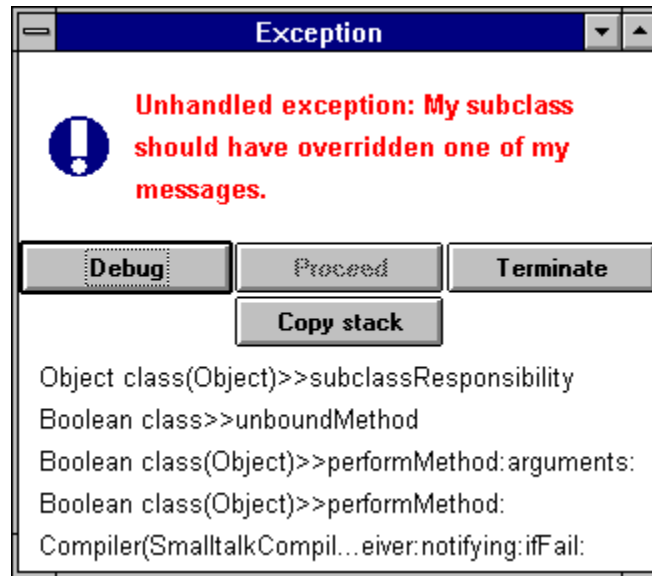


Figure 4.7. Result of trying to execute a message whose definition is left to a subclass.

The working definitions of `ifTrue:` are thus found in classes `False` and `True`. This is natural because class `Boolean` is abstract, no instances of it ever exist, and its definition of `ifTrue:` should thus never be executed.

We will now move on to subclasses, starting with `False`. In class `False`, message `ifTrue:` should *not* execute the alternative block and the definition is as follows:

`ifTrue: alternativeBlock`

"Since the condition is false, answer the value of the false alternative, which is `nil`. This method is typically not invoked because `ifTrue:/ifFalse:` expressions are compiled in-line for literal blocks."

`^nil`

In other words, the false object ignores the argument and returns the object `nil`. In class `True`, message `ifTrue: aBlock` must evaluate `aBlock` and the definition is thus

ifTrue: alternativeBlock

"Answer the value of alternativeBlock. This method is typically not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."

^alternativeBlock value

This definition sends message value to alternativeBlock which forces the block to execute the statements inside the square brackets. The caret then returns the value calculated by the last executed statement.

Message value is very important and it is thus useful to give an example to explain how it works. As an illustration of the operation of the value message

[Transcript clear; show: 'Testing block closure'; cr. Dialog warn: 'A test'] *value*

evaluates the statements inside the block and returns the result. It has exactly the same effect as

Transcript clear; show: 'Testing block closure'; cr. Dialog warn: 'A test'

If this is so, why should we ever want to put statements in a block and send them value? The reason is that we sometimes cannot predict which statements we will want to evaluate – and message ifTrue: is an example of this: We know that if the receiver is true, the method will want to evaluate a block of statements but these statements can be anything so the best we can do is put them inside a block and evaluate the block with value.

In-line messages

The comment of ifTrue: contains a note about *in-line compilation*:

"This method is typically not invoked because ifTrue:/ifFalse: expressions are *compiled in-line* for literal blocks."

What does this mean? Messages ifTrue: and ifFalse: are among the most frequently used messages and because they are used so often, they must be implemented very efficiently. To achieve this, when the compiler encounters ifTrue: or ifFalse:, it does not create code to send the messages in the definition as it normally does, but inserts code to execute the sequence of statements in the block directly. This technique is used by a very small number of Smalltalk methods and is called in-lining. In the Debugger, in-lined messages are marked as *optimized*.

You might be wondering why Smalltalk bothers with the definition of ifTrue: if it is not really used. There are two reasons for this. The less important one is that the definition shows the exact effect of the message. The more important reason is that there are special situations in which the compiler does *not* create in-line code (because it cannot) and executes the message using the definition instead. We will explain how these situations arise later.

Objects true and false are unique

True and False are among the very few classes that allow only one instance; class UndefinedObject is another one, and small integers, symbols, and characters also have this property. Such unique instances are sometimes called singletons. The single instance of True is called true, the single instance of False is called false, and the single instance of UndefinedObject is called nil. When the compiler reads Smalltalk code, it recognizes these special words called *literals* and treats them differently. This means that when you need a true object (or a false or a nil), you can specify it by using the literal form true (or false or nil) directly in the code. In other words, you don't have to create, for example, instances of True by sending

True new

or a similar message. In fact, this statement would not work and would open an Exception Window telling you that you cannot create new True objects. As we mentioned, there are a few other kinds of literals such

as literal strings such as 'This is a your tax', literal numbers such as 132 or 3.14, and literal blocks such as [Transcript cr].

Main lessons learned:

- A singleton is an instance of a class that allows only a single instance to exist. Examples of singletons are objects true, false, and nil.
- A literal is a textual representation directly converted into an object by the compiler. Examples of literals include numbers, strings, nil, true, false, and block literals.
- When the compiler encounters a message that is compiled in-line, it does not execute the messages in the body of the method but creates machine code instead. Smalltalk uses in-lining to increase operation speed in a few very frequently used messages.

Exercises

1. Find all definitions of ifFalse: using the *Browse implementors of* command and explain how they work.
2. Explain the definition of the new message in classes Boolean, True, and False.

4.4 Selecting one of two alternative actions

In the previous section, we dealt with situations where the program must decide whether to execute a block of statements or not. In this section, we will deal with situations in which the program must select one of two alternative actions.

Example 1: Definition of max:

Class Magnitude defines the properties shared by all objects that can be compared. If two objects can be compared, we can find which one is larger and which one is smaller and Magnitude thus contains the definitions of methods max: and min: which can be used as follows:

```
3 max: 4      "Returns 4."  
3 min: 4      "Returns 3."
```

The definition of max: is as follows:

max: aMagnitude

```
"Answer the receiver or the argument, whichever has the greater magnitude."  
self > aMagnitude  
  ifTrue: [^self]  
  ifFalse: [^aMagnitude]
```

Example 2. Decide whether three numbers form a right-angle triangle

Problem: Write a code fragment to prompt the user to enter three numbers, test the numbers, and print a message in the Transcript to say whether the numbers form the sides of a right-angled triangle or not.

Solution: If a, b, and c are the sides of a right angle triangle and c is the hypotenuse, then $c^2 = a^2 + b^2$. The algorithm for finding whether three numbers for a right-angle triangle is thus as follows:

1. Ask the user to enter the hypotenuse; call it c.
2. Ask the user to enter the next side; call it a.
3. Ask the user to enter the next side; call it b.
4. Clear the Transcript.
5. Check whether $c^2 = a^2 + b^2$.
6. If true, print 'triangle is right-angled'; if false, print 'triangle is NOT right-angled'.

To write the program, we need to know how to read a *number* from the keyboard and how to select one of two choices depending on the value of a Boolean receiver.

To read a number from the keyboard, read a *string* using `request:initialAnswer:`, and convert it into a number using the conversion message `asNumber`. To choose one of two alternatives use the Boolean message `ifTrue:ifFalse:`. Beginners often think that `ifTrue:ifFalse:` are two messages; in reality, it is a *single* message with two keywords, each expecting a block argument. Its operation is described in Figure 4.8.

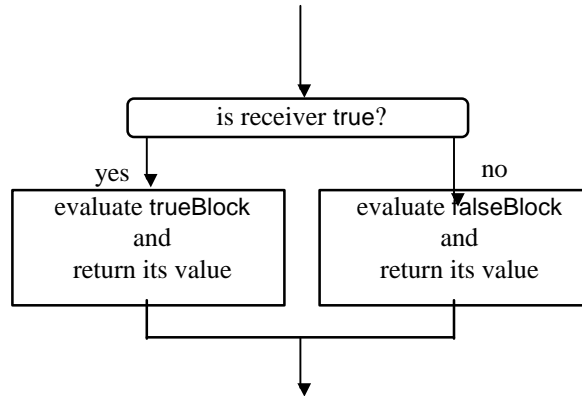


Figure 4.8. Evaluation of `ifTrue: trueBlock ifFalse: falseBlock`.

With this information, we can write the following first version of the program. We will see that this version of the program is not entirely correct but since the mistake is frequent and produces behavior that seems very confusing to a novice, we will start with the incorrect version.

```
|a b c|
c := Dialog request: 'Enter the length of the hypotenuse' initialAnswer: '' asNumber.
a := Dialog request: 'Enter the length of the second side' initialAnswer: '' asNumber.
b := Dialog request: 'Enter the length of the third side' initialAnswer: '' asNumber.
Transcript clear.
a squared + b squared = c squared
  ifTrue: [Transcript show: 'This IS a right-angled triangle']
  ifFalse: [Transcript show: 'This is NOT a right-angled triangle']
```

When we try to execute this program, Smalltalk opens the Exception Window in Figure 4.9. The problem looks very unpleasant because the displayed messages on the message stack don't look like anything in our code!

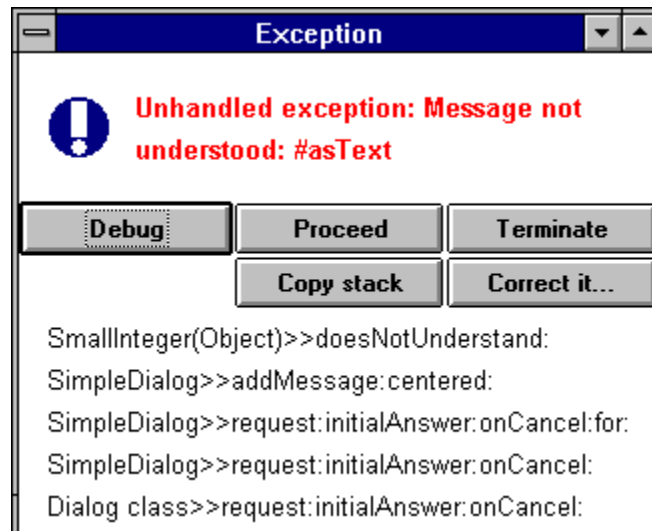


Figure 4.9. Exception Window produced by the first solution of Example 1.

We open the debugger to check where the problem occurred and find that our program is trying to execute message

request: 'Enter length of the first side - the hypotenuse' initialAnswer: '' asNumber

and fails.

Let's take a closer look at the evaluation of this line: The expression does not contain any parenthesized expressions. There is one unary messages - asNumber and the receiver is the empty string ''. The first thing that happens when the statement is executed is thus that Smalltalk takes an empty string and converts it to a number. This works and returns 0. (Method asNumber first initializes a temporary variable to zero and then keeps converting successive characters to numeric values until it reaches a character that is not a digit. It then stops and returns the result. In our case, there are no digits and the returned result is thus 0.) Our expression now effectively becomes

request: 'Enter length of the first side - the hypotenuse' initialAnswer: 0

This expression does not contain any more unary messages. There are no binary messages but there is a keyword message request:initialAnswer:. Its first argument is a string and the second argument is a number. Smalltalk starts executing this message and fails because the definition of request:initialAnswer: assumes that *both* arguments are strings. Now the message in the Exception Window makes more sense: It essentially says that an integer does not how to convert itself to a Text object.

To correct the problem, think about our goal: We first want to read a string using request:initialAnswer: and *then* convert it to a number. The correct formulation is thus

a := (Dialog request: 'Enter length of the first side - the hypotenuse' initialAnswer: '') asNumber.

Since our code fragment uses the same pattern throughout, we must modify the rest in the same way:

```
|a b c|
Dialog warn: 'You will be asked to enter three sides of a triangle. The first number is the hypotenuse'.
c := (Dialog request: 'Enter length of the first side - the hypotenuse' initialAnswer: '') asNumber.
a := (Dialog request: 'Enter length of the second side' initialAnswer: '') asNumber.
b := (Dialog request: 'Enter length of the third side' initialAnswer: '') asNumber.
Transcript clear.
a squared + b squared = c squared
  ifTrue: [Transcript show: 'This IS a right angled triangle']
  ifFalse: [Transcript show: 'This is NOT a right angled triangle']
```

This works but a note of caution is in order. When a condition is based on testing numbers, such as $c^2 = a^2 + b^2$, we will get the exact result when using integers (such as 3, 4, and 5) but we cannot be certain that the test will be evaluated accurately if the numbers are floating-point numbers such as 3.14 or -0.45. Computer arithmetic with floating-point numbers always performs conversion to binary and this conversion is almost always inaccurate. Numeric comparisons involving floating-point numbers should thus be avoided and if we must compare floating-point numbers, we should accept a tiny difference as equality. This problem is common to floating-point arithmetic in any language, not only in Smalltalk.

Example 3: Calculating the value of a function with a complicated definition

Problem: Obtain a number from the user and calculate and print the argument and the value of a function defined as follows: If the argument is negative, the result is 0. If the argument is greater or equal to 0 and less or equal to 10, the result is equal to the original argument. If the argument is greater than 10 and less than 100, the result is the square of the argument. For all other argument values, the result is the cube of the argument. Print the original argument and the result in the Transcript. Sketch the graph of the function.

Solution: The solution consists of a sequence of consecutive tests directly reflecting the definition:

```
| argument result |
argument:= (Dialog request: 'Enter a number' initialAnswer: '') asNumber.
argument < 0
    ifTrue: [result := 0]
    ifFalse: [argument <= 10
        ifTrue: [result := argument]
        ifFalse: [argument < 100
            ifTrue: [result := argument squared]
            ifFalse: [result := argument * argument * argument]]].
"Print the argument and the result."
Transcript clear;
show: 'Argument ', argument printString;
tab;
show: ' Result ',result printString
```

This example required nesting the tests one inside the other and in such a case, matching of brackets is sensitive. If you are not sure whether your brackets match, click twice just after an opening bracket or just before a closing one and the text between the brackets will be highlighted.

Example 4: The definition of ifTrue:ifFalse:

The definition of ifTrue:ifFalse: in class True must evaluate the first block argument and ignore the second and this is exactly what happens:

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

"Answer the value of trueAlternativeBlock. This method is typically not invoked because ifTrue:/ifFalse: expressions are compiled in-line for literal blocks."

^trueAlternativeBlock value

Can you use this model to write the definition of ifTrue:ifFalse: in class False without looking? The library also contains an ifFalse:ifTrue: method which has a similar effect and is included only as a very rarely used convenience.

Main lessons learned:

- To select one of two alternative blocks of statements, use ifTrue:ifFalse: or ifFalse:ifTrue:.
- Comparison based on floating-point numbers is almost always inaccurate and should be avoided.

Exercises

1. Write a code fragment to request a number, ask whether it should be negated, and print the result in the Transcript window. (Hint: Use unary message negated; putting a minus sign in front of a variable as in `x := -y` will not work - try it.)
2. Implement the multiple tax bracket problem from Section 4.1 and print the input, intermediate results, and total tax in the Transcript.
3. Display all references to ifTrue:ifFalse: and to ifFalse: ifTrue: and comment on your findings.
4. If you must compare two floating-point numbers, you must be prepared for a small inaccuracy. You can, for example, say that `x` and `y` are 'equal' if their difference is smaller than one millionth of the smaller of the two numbers. Modify Example 2 along these lines. (Hint: Use message min:.)

4.5 Use ifTrue: and ifFalse: only when necessary

Probably the most important rule of programming is that code should be as simple as possible and one of the most common violations of simplicity is the use of selection messages. In Smalltalk, fortunately,

you can usually use polymorphism instead of `ifTrue:/ifFalse:`. We have already mentioned this when we discussed polymorphism and we will now briefly restate the example.

Assume that we want to write a drawing program for drawing various geometric shapes. One possible solution is to define a `Pen` class with a drawing method `draw:at:` along the following lines:

draw: aGeometricShape at: aPoint

"Draw aGeometricShape at the specified point of the screen."

aGeometricShape isMemberOf: Line "Check whether the receiver's class is Line."

ifTrue: [aShape drawLine at: aPoint].

aGeometricShape isMemberOf: Rectangle

ifTrue: [aShape darwRectangle at: aPoint].

aGeometricShape isMemberOf: Circle

ifTrue: [aShape drawCircle at: aPoint].

aGeometricShape isMemberOf: FreeLine

ifTrue: aShape drawFreeLine at: aPoint].

etc.

With a reasonable number of geometric shapes, this method will be long and awkward. It will also be inefficient because drawing a shape will require testing chain of `ifTrue:` messages, wasting a lot of time just finding which part of the code should be executed. Moreover, adding a new shape will require adding code to this method and any other method that involves some kind of drawing, and forgetting to modify one of them is a distinct possibility. A much better solution is to define a special drawing method `drawAt:` for each kind of geometric object. The `Pen`'s drawing method is then simply

draw: aGeometricShape at: aPoint

"Draw aGeometricShape at the specified point of the screen."

aGeometricShape drawAt: aPoint

This is much simpler, much easier to understand, and faster. Besides, if a new kind of geometric object is added, this method does not have to be changed at all. But is it always possible to eliminate tests? Obviously not. If we want to ask the user whether to delete a file or not, we must post a yes/no answer and decide using `ifTrue:/ifFalse:`. Similarly, if we want to calculate tax in a multiple tax bracket system, we must test whether the numerical values are in certain ranges or not. There are many situations like this but if we can avoid `ifTrue:/ifFalse:` by using polymorphism, we always should.

Main lessons learned:

- Using `ifTrue:/ifFalse:` indiscriminately leads to complex and inefficient code that is difficult to modify.
- Use polymorphism instead of `ifTrue:/ifFalse:` whenever possible.

4.6 Creating a new class and a new method

Up to this point, we have been experimenting with small code fragments but programming in Smalltalk consists of creating classes and methods. Now that we know what a method looks like, let's write one.

Example 1: Class Name and its creation, initialization, and accessing methods

Although this example does not have anything to do with Booleans, we think that it is useful because it shows how to create classes and methods on the simplest possible example. We will develop more relevant examples later.

Assume that we have an application that requires personal names and that we need to distinguish between first name, middle name, and last name. This means that we cannot group all parts of a name into a single string and to keep the parts separated, we must assign one instance variable to each. We will call these instance variables `firstName`, `middleName`, and `lastName`. With this, we are now ready to define the

class. To do this, open a Browser, and add a new category called for example Tests using the *add ...* command in the <operate> menu of the category view (Figure 4.10).

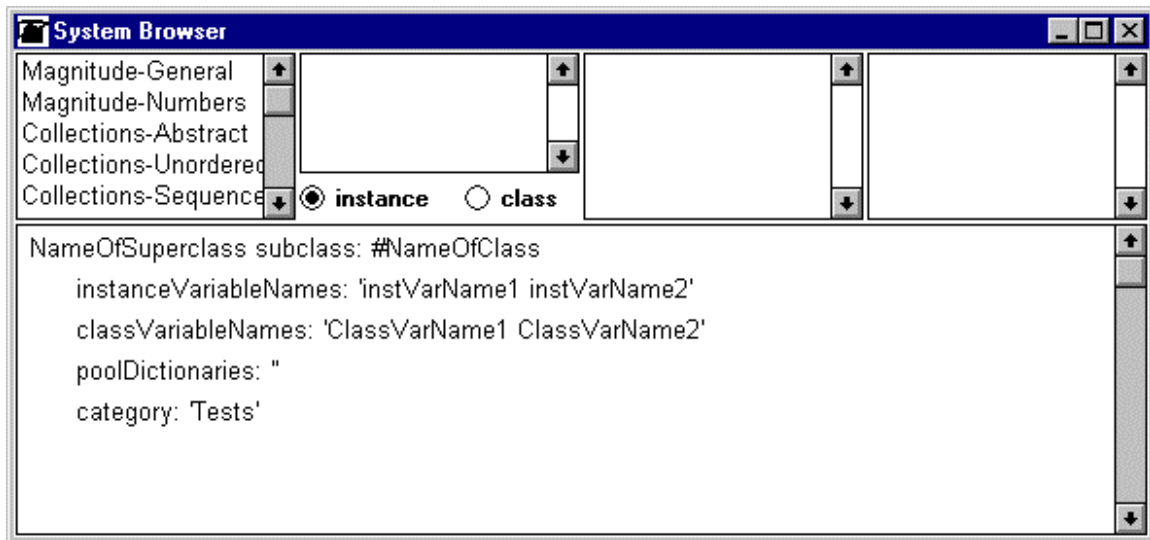


Figure 4.10. System Browser showing a class template.

The Browser now displays a template for creating a new class, actually a Smalltalk message whose execution will create the class. To complete the template, we must enter the name of the superclass (we will use Object because no related class is available), the name of the class (Name), names of instance variables (as decided above), and names of class variables (none) and poolDictionaries (none). Finally, click accept in the <operate> menu of the code and unless you made a typing mistake, the new class is added to the library. The resulting browser showing the completed template and the new class is as in Figure 4.11.

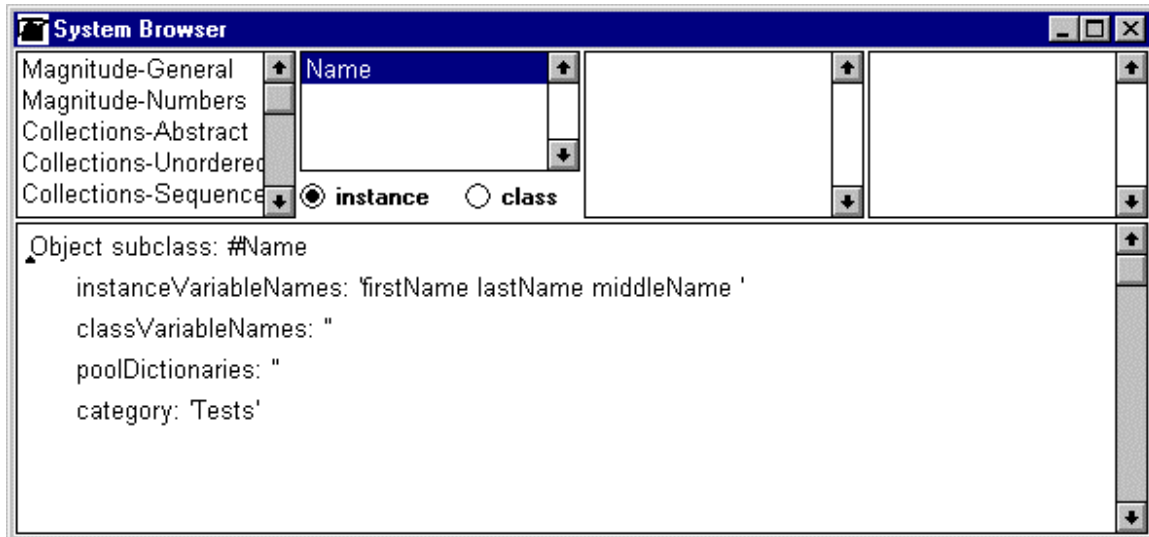


Figure 4.11. Browser after adding class Name.

The next step is to add class comment and to do this, click comment in the class view's <operate> menu. The bottom view of the browser changes and displays the following text:

This class has not yet been commented. The comment should state the purpose of the class, what messages are subclassResponsibility, and the type and purpose of each instance and class variable. The comment should also explain any unobvious aspects of the implementation.

Select the whole text and replace it with a comment. There are two styles for writing the comment. One is very antropomorphic and uses the first person as in 'I represent a name ...'. Its advantage is that it makes you think in terms of the class which always helps. The other style is less personal and uses the third person as in 'This class represents a name ...'. This style seems less extravagant. Choose the style that you like better and use it consistently. We will use the first style because it will force us to think as if we were the class which seems a good idea at this point of the book. Following the template, we write

I represent a personal name consisting of a first name, a middle name, and a last name.

Instance variables

firstName	<String>	first name
middleName	<String>	middle name
lastName	<String>	last name

The comment seems almost unnecessary because it seems very obvious but it is an essential practice to use class comments. As an example, the indication that the values of variables should be String objects is important.

At this point, open a workspace and execute

Name new

with *inspect*. Method new is inherited by all classes from class Behavior and it creates a new instance of the class with uninitialized instance variables. Obviously, this is not good enough and we must now create methods to assign values to these instance variables, and to access the values once the object exists (because the values of instance and class variables can only be accessed by instance or class messages). Such messages are called *accessing* messages and they come in two varieties which are often called *get messages*, and *set messages*. Get messages return the value of the variable, set messages set its value to a new object. We will thus define accessing methods for all our variables and put them into a protocol called accessing. (The most common kinds of protocols have established names which are listed in Appendix 4.) To do this, use command *add ...* in the <operate> menu in the protocol view of the browser. You will get the template in Figure 4.12. The template explains the basic structure of a method (as we already explained) and to create a method, simply select all the text and start typing the definition of the method.

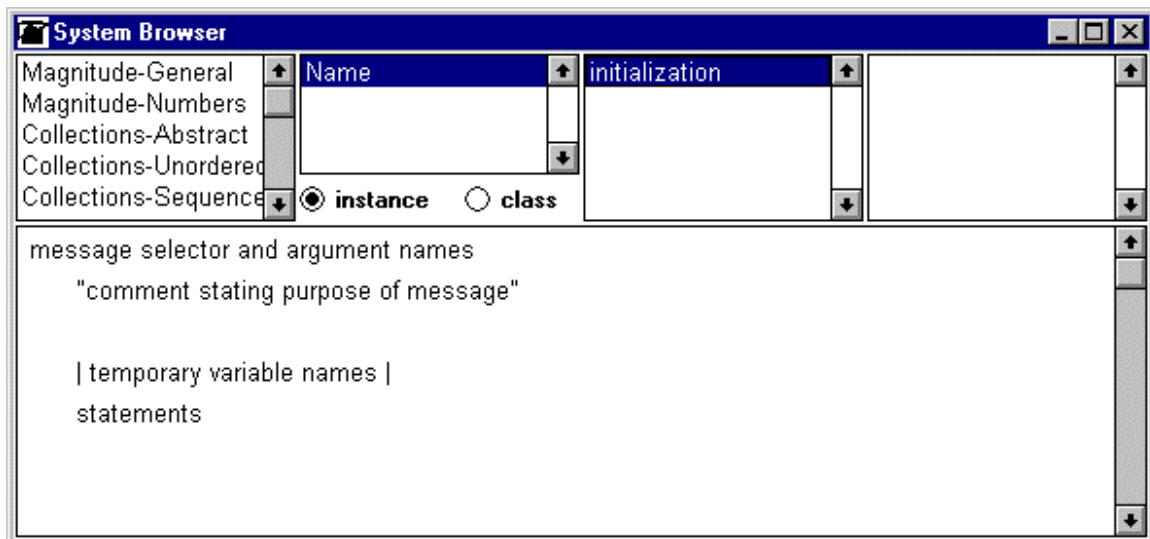


Figure 4.12. Browser with method template.

Let's start with the get method for the `firstName` instance variable. A get message simply returns the value of a variable and it is standard practice that it is named with the name of the instance variable. The definition is thus

```
firstName  
  ^ firstName
```

Accessing methods are so common and so simple that programmers don't normally comment them.

After typing the method, select the *format* command in the <operate> menu of the code view and the code will be automatically formatted. If you made syntax mistakes, you will get a warning and you will have to correct your mistake. After this, execute the accept method which will compile the code and add the method to the library (Figure 4.13).

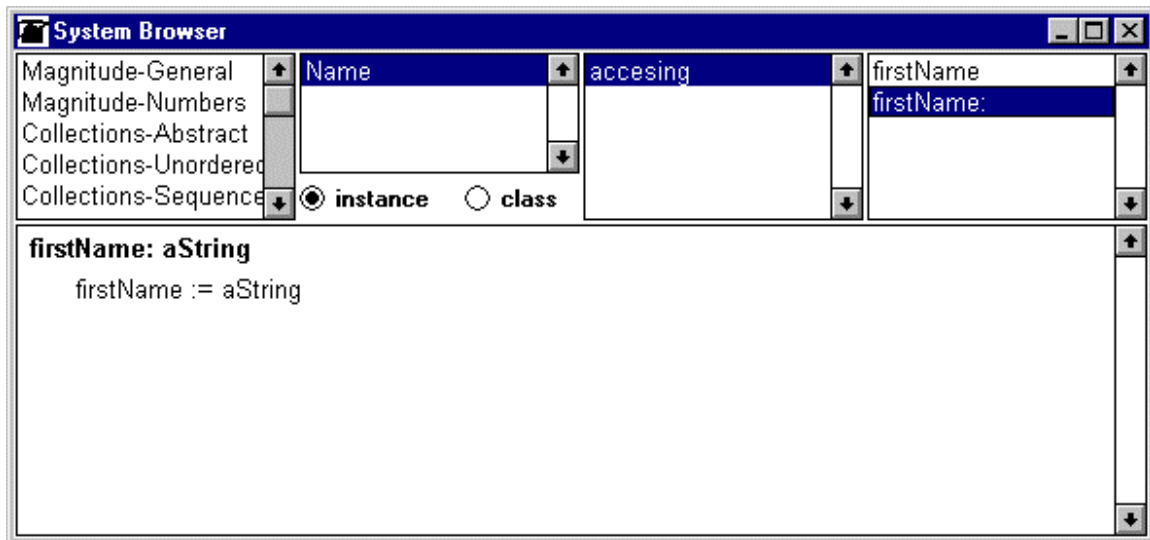


Figure 4.13. Browser showing the new accessing method.

We will now write the corresponding set method and leave it to you to define the remaining get methods. A set method simply assigns a new value to an instance variable and the new value is supplied as the argument. Our set method is thus a keyword method:

```
firstName: aString  
  firstName := aString
```

We leave it to you again to define the remaining set methods and proceed to a little test. We should now be able to create a new `Name` object and assign values to its instance variables. As an example,

```
Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith'
```

should create a `Name` corresponding to John Mathew Smith. Assuming that you created the remaining set accessors, type this code into a workspace and execute with inspect. When you examine the instance variables, you will find that they indeed have the desired values.

As our next step, we will test our get methods. As an example,

```
(Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith') firstName
```

should return 'John'. Type this expression into a workspace and execute with *print it* to see that it indeed does.

Now that we think about it, it seems that our approach could cause a problem. A programmer might, for example, create a Name object and give it a first and last name but no middle name. If a program then tried to print the name, it would fail on the middle name because its value would be nil and nil does not understand String messages. It will thus be a good idea to modify the instance creation message so that it initializes all instance variable to empty strings with no characters in them. This is, of course, rather useless but at least it will not cause a crash if a variable is left unassigned.

There are several possible solutions to our problem and the simplest one is to define a new initialization method that forces the user to enter some strings for each of the instance variables as in

Name first: 'John' middle: 'Mathew' last: 'Smith'

This new message is obviously a class message because the receiver is the class which creates the object. We will thus define it on the class side and put it into a new protocol called instance creation . Its execution consists of creating an instance using new, and initializing it with accessing messages as we did above. The definition is thus

first: firstString middle: middleString last: lastString

"Create and return an initialized instance."

^(self new) firstName: firstString; middleName: middleString; lastName: lastString

A few notes are in order:

1. If we did not include the ^ symbol, the method would return the receiver (the *class* Name) rather than the created *instance*. This is a typical omission that frequently creates problems.
2. We could have used Name instead of self with the same result because the receiver of this message (object self) is class Name. However, this could create an undesirable effect if we subclassed Name and the style shown above is thus the preferred one.
3. We could have left out the brackets around self new and get the same result: Message firstName: message would go to the new instance of Name, and the remaining two messages would go to the receiver of firstName:, in other words the new instance of Name (now modified by message firstName:).

Class Name is now complete and we leave it to you to test that our new instance creation message indeed works as desired.

Example 2. Test of asUppercase

We have already mentioned that all code must be carefully tested. To test a new class, we usually write methods that test the essential methods of the class and add them to the class protocol. Even better, we could write a special class containing the tests. In this example, we will use the first approach and write a simple method to test the asUppercase method.

Although there is no general rule for writing test methods, the principle is that we find some situations where the tested method should do useful work, and some situations where it should not do anything and test both. We must test both typical and 'boundary' situations.

In our case, asUppercase should convert all lowercase letters to uppercase but it should not affect any other characters. In particular, it should not change any uppercase letters, digits, punctuation symbols, and spaces. The method should work on strings consisting of lowercase letters only, as well as strings containing lowercase letters and other characters.

Our test method will work as follows: For each tested situation, we will construct a string, send it the asUppercase message, and compare the returned and the expected result. We will then write a message reporting the success or failure of the test to the Transcript. As an example, the first test might be

```
'abcd' asUppercase = 'ABCD'  
  ifTrue: [Transcript show: 'test 1 passed']  
  ifFalse: [Transcript show: 'test 1 failed']
```

We will add the test method to class `CharacterArray` because that's where `asUppercase` is defined, and define it in a *class* protocol. Since no suitable class protocol exists, we will create a new one called `testing`. The definition of the method will be as follows:

testConversion

"Test asUppercase conversion on several selected receivers."

```
Transcript clear.  
'abcd' asUppercase = 'ABCD'  
    ifTrue: [Transcript show: 'test 1 passed']  
    ifFalse: [Transcript show: 'test 1 failed'].  
Transcript cr.  
'ABCD' asUppercase = 'ABCD'  
    ifTrue: [Transcript show: 'test 2 passed']  
    ifFalse: [Transcript show: 'test 2 failed'].  
Transcript cr.  
'abCd' asUppercase = 'ABCD'  
    ifTrue: [Transcript show: 'test 3 passed']  
    ifFalse: [Transcript show: 'test 3 failed'].  
Transcript cr.  
'123' asUppercase = '123'  
    ifTrue: [Transcript show: 'test 4 passed']  
    ifFalse: [Transcript show: 'test 4 failed'].  
Transcript cr.  
'ab12?Cd' asUppercase = 'AB12?CD'  
    ifTrue: [Transcript show: 'test 5 passed']  
    ifFalse: [Transcript show: 'test 5 failed']
```

To add the method to the library, open a System Browser on class `CharacterArray`, create a testing protocol on the class side using the technique explained in Example 1, and enter, format, and accept the new method.

Once the method is compiled, we must test that it indeed works. To perform the test, send the test message to class `CharacterArray` (or to any of its subclasses) as in

`CharacterArray testConversion`

The message prints confirmations that all tests succeeded into the Transcript view.

Assume now that this is a convenience method and that we don't want to clog the library with it. We will thus save its code in a file and remove the method from the library. To do this, open the `<operate>` menu in the method view of the System Browser and select command *file out as ...* (Figure 4.14).

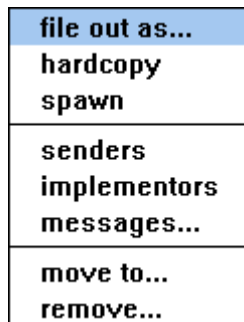


Figure 4.14. `<operate>` menu command for filing out a method. A similar menu is available in other System Browser views to file out a whole protocol, a whole class, or a whole category of classes.

The command opens a dialog requesting the name of the file as in Figure 4.15. Accept the proposed name or enter another name and possibly a new directory path, and click *OK*. If you did not specify a new directory path, the file (a *fileout* of the code) will be stored in the *image* sub-directory of your VisualWorks directory.

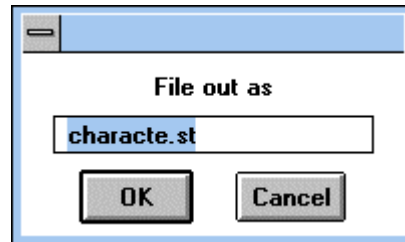


Figure 4.15. The fileout dialog with the default filename. The default directory is *image*.

Now that we saved the method, we can delete it and its protocol from the library using the *remove* command in the <operate> menu of the protocol view. If you ever want to restore the protocol and the method (to test *asUppercase* after some changes), open the File Editor on the file (Figure 4.16) and file the code in using the *file in* command of its <operate> menu. This reads the file and restores its contents (method, protocol, class, or category) in the library. This happens because fileout files use a special format and as they are filed in, the compiler automatically translates them back into the library.

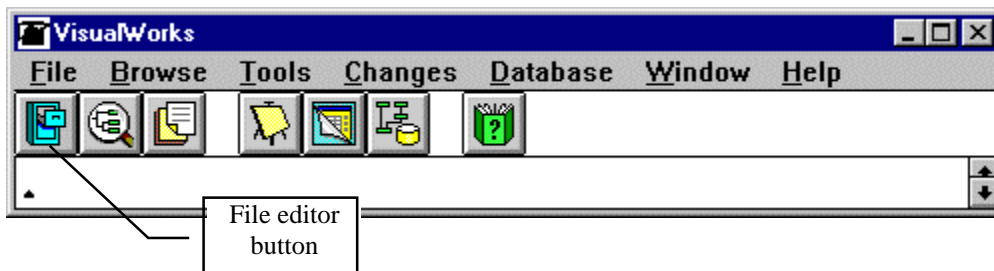


Figure 4.16. To open a File Editor, use the leftmost button or the *File Editor* command in *Tools*.

Main lessons learned:

- Use the System Browser to create new categories, classes, protocols, and methods.
- Testing is usually implemented by adding test methods to the class being tested or by creating a special test class.
- Collect test methods in a testing protocol on the class side.
- Remove test methods from the library when not needed but save them in a file first.
- To save a method, a protocol, a class, or a category in a file, file it out. To restore it, file it in.
- Fileout files use a special format and the compiler restores the contents into the library during fill-in.

Exercises

1. Implement the examples in this section.
2. It would seem that instead of typing
(Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith') firstName
we could have used
Name new firstName: 'John'; middleName: 'Mathew'; lastName: 'Smith' firstName
with exactly the same result. Is that correct?
3. Protocol converting in *CharacterArray* contains several other conversion methods. Extend our test method to test methods *asLowercase* and *asNumber*.

4. Write and execute a test method to test some of the methods in CharacterArray protocol *comparing*. Include tests of methods < and > (these methods compare strings on the basis of their collating sequence - basically their alphabetical order) and match: which compares strings using wildcard characters (* matches any string, # matches any single character). The comment of match: includes examples demonstrating the purpose and use of the method. Including examples in method definition is a common practice.
5. Write and test method requestNumber: aString initialAnswer: aNumber. The method should work just like request:initialAnswer: but return a number. (Hint: Use the principle that we used in Section 4.4.) What is the best class for the method?
6. Define and test method cubed to calculate the third power of a number. Use squared as a model.
7. Define class Address with instance variables street, number, city, postalCode. The values of all instance variables are strings. Define all necessary methods and test them.
8. Define class RentalProperty with instance variables address (an instance of Address from Exercise 7) and numberOfApartments (an integer). Define all necessary methods and test them.

4.7 Logic operations

A test condition often has several components. As an example, when we want to check whether a number is greater than 5 and divisible by 7, we must combine two tests into one. Operations combining true and false objects are called *logic operations* and the Smalltalk library provides several methods to implement them.

The most common logic operations are called *not*, *and*, and *or*. Although the names are derived from English and make a lot of sense, their interpretation could be ambiguous and the definition is thus better expressed by tables. Because the tables deal with true/false values, they are usually called *truth tables*. The truth tables of *not*, *and*, and *or* are given in Table 4.1 and you can check that their meaning corresponds to the meaning of the words *not*, *and*, and *or* in everyday communication. As an example, if *a* is true, *not a* is false, and so on.

a	not a
false	true
true	false

a	b	a and b
false	false	false
false	true	false
true	false	false
true	true	true

a	b	a or b
false	false	false
false	true	true
true	false	true
true	true	true

Table 4.1. Truth tables of (left to right) *not a*, *a and b* (conjunction), *a or b* (disjunction).

In Smalltalk, not, and, and or are defined in classes Boolean, True, and False and used as follows:

```
aBoolean not
aBoolean and: ["statements that evaluate to true or false"]
aBoolean or: ["statements that evaluate to true or false"]
```

They all return a Boolean result and we will now illustrate them on several examples.

Example 1. Test whether an income is taxable

Problem: Write an expression that tests whether a tax payer's income is taxable or not. The rules are as follows: Income is taxable if the person is up to 60 years old and the income is greater than \$25,000, or the person is over 60 and the income is greater than \$30,000.

One possible solution is to evaluate the two rules separately, store the results in variables, and combine the variables. This can be expressed as follows:

```
| age income rule1 rule2 |
age := (Dialog request: 'Enter your age' initialAnswer: '') asNumber.
income := (Dialog request: 'Enter your income' initialAnswer: '') asNumber.
```

```
rule1 := age <= 60 and: [income > 25000].  
rule2 := age > 60 and: [income > 30000].  
(rule1 or: [rule2])  
    ifTrue: [Dialog warn: 'Tax is payable']  
    ifFalse: [Dialog warn: 'No tax payable']
```

Another possibility is to eliminate the rule variables and combine all the rules into a single calculation as follows:

```
| age income |  
age := (Dialog request: 'Enter your age' initialAnswer: '') asNumber.  
income := (Dialog request: 'Enter your income' initialAnswer: '') asNumber.  
((age <= 60 and: [income > 25000]) or:  
 [age > 60 and: [income > 30000]])  
    ifTrue: [Dialog warn: 'Tax is payable']  
    ifFalse: [Dialog warn: 'No tax payable']
```

This code has exactly the same effect but it is harder to read and more error prone. We prefer the first solution.

Example 2. Safeguarding against an illegal message

The fact that the argument of and: and or: is a block is important because a block is not just a group of statements but rather a group of statements whose evaluation is deferred until explicitly requested by a value message. In the case of and: and or: this allows us to safeguard against an illegal message. As an example, consider the following fragment:

```
...  
(x >= 0 and: [x sqrt <= 5])  
ifTrue: [Dialog warn: 'Argument ', x printString, ' is legal']  
ifFalse: [Dialog warn: 'The argument is illegal']  
...
```

This code fragment works for any value of x because it evaluates the block argument of and: only if the first test succeeds and defers (and never executes) it if the test fails. If this were not so and if the value of x was negative, the program would crash when attempting to execute the statement in the block argument because negative numbers crash on the sqrt message.

Example 3: Use of and: in the definition of between:and:

Method between:and: returns true if the receiver is greater or equal to the first argument and less or equal to the second argument:

3 between: 1 and: 5 "Returns true."

whereas

3 between: 10 and: 50 "Returns false."

Its definition in class Magnitude is as follows:

```
between: min and: max  
    ^(self >= min) and: [self <= max]
```

The method takes the receiver and sends it the message >=, getting a true or false result. If the result is false, it does not evaluate the block because the result must be false (see Table 4.1). If the result is true, it sends and: with

`[self <= max]`

to the true object. This returns true or false. The Boolean result is then returned. Note that all subclasses of Magnitude including Date, Time, Character, and all number classes inherit this method.

Example 4: A ticketing program

Problem: A computer program issues tickets in a museum and calculates the price on the following rule: A ticket is \$1 on weekdays between 9 a.m. and 12 a.m., on Saturday after 12:00, and any time on Sunday. The rate is \$2 at all other times.

Solution: The essence of the solution is as follows:

1. Get current time.
2. Get today's day of the week.
3. Use the time and date to determine which rate applies.

To write the program, we need to know how to find the current time and day of the week. We suspect that there are classes and messages to do this and we indeed find classes `Time` and `Date`.

When we check the instance methods of class `Time`, we don't find anything that appears suitable for our purpose. We thus open class protocols and find method called `now` in the instance creation protocol. Its comment says "Answer a `Time` representing the time right now--this is a 24 hour clock". To test this method, we execute

`Time now`

with *inspect*, and get a `Time` object with components `hours`, `minutes`, and `seconds`. Hours are counted from 0 to 23, minutes and seconds are counted from 0 to 59. Our next question is how to extract the hours component. `Time now` returns an instance of `Time` and we thus check instance methods of `Time` at the instance methods side; in the accessing protocol, we find method `hours`. When we execute

`Time now hours`

with *print it*, we get just what we wanted.

Now that we know how to get time, we search class `Date`. We expect that the method returning today's date will be a class method, probably in the instance creation protocol and we indeed find a method called `today` which seems to do the job. Executing

`Date today`

with *inspect*, returns a `Date` object with two instance variables called `days` and `year`. We are interested in days of the week but is that what does `days` mean? When we open the comment on `Date` we find that `days` is the number of days since the start of the year. Is there some way to convert this number into the day of the week? If there is, it must be an instance method because `Date today` is an instance of `Date`. We thus look under the instance methods of `Date` and find method `weekday` that seems to do exactly what we need. When we try

`Date today weekday`

it returns a string such as 'Thursday'. We now have all the ingredients and we can write the code which uses heavily the `or:` method:

```
| dayToday price |
"Get time and day now."
timeNow := Time now hours.
dayToday := Date today weekday.
"Find which rate applies."
(((dayToday = 'Sunday')
 or: [dayToday = 'Saturday' and: [timeNow >= 12]])
 or: [timeNow between: 9 and: 12])
 ifTrue: [price:= 1]
 ifFalse: [price:= 2].
Transcript clear; show: 'price is ', price printString
```

Example 5: Multiple logic operations

Problem: Get user's gender, age, and weight. If the user is male, over 50 years of age, and weighs 150 kg or more, display a warning saying 'You should take a look at your diet'.

Solution: To obtain the information, we will use multiple choice dialogs. These dialogs are easy to construct - the one in Figure 4.17 was obtained with the following code:

Dialog	choose: 'What is your gender?'	"Prompt message."
	labels: (Array with: 'female' with: 'male')	"Labels on buttons."
	values: (Array with: 'female' with: 'male')	"Objects returned when a button is clicked."
	default: nil	"Specifies initially selected button."

The code uses some concepts that we have not yet covered but the pattern is obvious and we will use it without further questions.



Figure 4.17. A multiple choice dialog for Example 3.

The logic of the problem requires that we combine two and: messages and to do this, we must nest the second and: into the block argument of the first and:. The complete solution is as follows:

```
| gender overFifty largeWeight |
"Get the information."
gender := Dialog choose: 'What is your gender?'
               labels: (Array with: 'female' with: 'male' )
               values: (Array with: 'female' with: 'male' )
               default: nil.
overFifty := (Dialog confirm: 'Are you at least 50?').
largeWeight:= Dialog confirm: 'Is your weight 150 kg or more?' .
"Evaluate collected information and output warning if appropriate."
((gender = 'male') and: [overFifty and: [largeWeight]])
  ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

How are logic operations implemented?

You might expect that the definition of and: takes the receiver and the argument and uses the truth table to decide whether to return true or false. In fact, it does not because there is a better way. In class False, and: simply returns false (the receiver)

and: alternativeBlock

```
"Don't bother evaluating the block - according to the truth table, the result must be false."
^self
```

because the truth table of and shows that when the receiver is false, the result must also be false.

We have already mentioned that the fact that this definition eliminates evaluation of the block argument may be used as a safeguard. It can also save a substantial amount of time because the calculation could be very time consuming as in

```
30 between: 5 factorial and: [30000 factorial / 256000.13 squared]
```

and is not needed if we only want to know whether the result is true or false. This approach is known as *non-evaluating conjunction* or *short-circuited conjunction* because it eliminates evaluation of the argument

block when it is not necessary to obtain the true or false result. The `or:` message is implemented in a similar way.

The definition of `and:` in class `True` is also based on the truth table. When you check what is the value of `and` when the receiver is true, you will find that the result is always the same as the argument. The definition in class `True` thus simply evaluates the argument block and returns it:

and: alternativeBlock

"Evaluate the block because the result might be true or false. The value is the same as the block's value."
^alternativeBlock value

Fully evaluating *and* and *or*

Non-evaluating *and* and *or* may not be acceptable when the argument block does something that must not be skipped. As an example, assume that you want to draw two pictures and check whether both fit within some rectangle R. A possible approach is

"Draw the first picture and calculate its bounding box (the rectangle containing it). Check whether the bounding box is included in rectangle R."

and

"Draw the second picture and calculate its bounding box. Check whether the bounding box is included in rectangle R."

"Return the result (true or false)."

If we implemented this logic using `and:` and if the first bounding box was not in rectangle R, the block that draws the second picture would not be evaluated and the second picture would not be drawn. This may not be what we wanted because the *side-effect* of the block (the drawing of the rectangle) may always be desired. For situations such as these, Smalltalk provides *fully-evaluating and* and *or* denoted `&` and `|`. Fully-evaluating logic always evaluates both arguments even when this is not required by the truth table. We will see below that both of these messages take a Boolean rather than a block as their argument.

If we have a choice, non-evaluating logic is better because it may be faster (when the evaluation of the argument is not required, it is skipped) and safer. It also provides the option of exiting from a method by using a `^` return operator in the block. Fortunately, we can always eliminate fully evaluating logic because there is always a way to solve a problem without side-effects. As an example, a better way to solve the above problem is as follows:

"Draw the first picture and calculate its bounding box box1."

"Draw the second picture and calculate its bounding box box2."

"Check that box1 and box2 are within R using *non-evaluating* conjunction."

Stylistically, fully-evaluating logic also has its advantage – it can make the code easier to read. As an example, the nested conditions in Example 3

```
((gender = 'male') and: [overFifty and: [overweight]])  
  ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

could be implemented in a more readable fashion with `&` as follows:

```
(gender = 'male') & overFifty & overweight  
  ifTrue: [Dialog warn: 'You should take a look at your diet.']
```

with the same effect. If the run-time penalty is acceptable, there is nothing wrong with this approach.

Main lessons learned:

- Logic operations such as *not*, *and*, and *or* are defined by truth tables derived from natural language.
- Logic *and* and *or* can be implemented as fully-evaluating or non-evaluating operations. Fully evaluating logic always evaluates both the receiver and the argument, non-evaluating logic evaluates the argument only if it is necessary.
- Non-evaluating (shortcut) logic with *and:* and *or:* speeds up execution and provides a means to protect against attempts to execute illegal messages.
- Non-evaluating logic *must* be used if the evaluation of the receiver is also a test whether the block argument should or should not be evaluated.
- Blocks represent deferred evaluation which means that the statements inside a block are ignored until their execution is explicitly requested.
- Smalltalk library contains built-in classes and methods to do almost anything. To learn Smalltalk, learn how to search the library.

Exercises

1. Write programs to solve the following problems:
 - a. Tax calculation: Tax is 0 if the person's income is less than \$10,000, or if the age is less than 18 years or more than 65 years. The tax is 10% of total income for all other income earners.
 - b. Student registration fee is \$5,000 unless the student's grade average is between 80 and 90 (inclusive) and the student lives in a residence (registration \$4,000), or the grade average is between 91 and 100 (registration fee \$2,000).
2. If you did the previous exercise, you found that when we need to check multiple conditions, the expression can get quite ugly and confusing because of the many brackets. Define two new methods *and: block1* and *or: block1* that allow two conditions to be packed into one message and make the code somewhat more compact and readable. Use the method to re-implement Exercise 1.
3. Correct the bracketing of the following expressions to make it syntactically correct. Assume that we are not using the methods from Exercise 2.
 - a. (square1 intersects: square2) and: [square1 includes: point1 and: square2 includes: point2]
 - b. x between: 3 and: y or: [(x > 1000 or: (x < -1000))]
4. Read and explain the definitions of *or:*.
5. Read and explain definitions of *&* and *|* - fully-evaluating conjunction and disjunction.
6. How many times are *and:*, *or:*, *|* and *&* used in the base library?
7. List five most useful methods of Date and Time.
8. Test the ticketing program with suitable combinations of days and times to make sure that its logic is correct. (Hint: Since we are testing the logic, it does not matter how the Date and Time objects are created. Use suitable alternative creation messages.)
9. Select a method in protocol testing in class Rectangle and explain how it works.
10. Predict the result of the evaluation of the following expressions, test, and explain:
 - a. (3 < 5) and: [6 squared]
 - b. (30 < 5) and: [6 squared]
 - c. (3 < 5) & [6 squared]
 - d. (30 < 5) & [6 squared]
 - e. \$a between: 7 and: \$z

4.8 Exclusive or, equality, and equivalence

Besides *not*, *and*, and *or*, another useful logic function is *exclusive or* (usually abbreviated as *xor*). Its definition is as follows:

x xor y

is true if x and y have different Boolean values, and false otherwise. In Smalltalk, *xor* is implemented as a Boolean keyword message *xor:* with one *Boolean* argument

aBoolean *xor:* anotherBoolean

As an example of the meaning of *xor*, assume that a restaurant customer ordering from a fixed menu must choose either a cake or an ice cream but not both. An expression to check that an order follows this rule could be written as

legalOrder := ordersCake *xor:* ordersIceCream

where *ordersCake* and *ordersIceCream* are true or false. If the expression returns true, the order is OK, otherwise, something is wrong.

Implementing *xor:* requires only checking whether the receiver and the argument are the same objects and then inverting the result using logical *not*. There is no need to have one definition for True and another for False, and *xor:* is thus defined in class Boolean as

xor: aBoolean

^(self == aBoolean) not "The == message tests equivalence."

Classes True and False inherit this definition.

The definition of *xor:* introduces the notion of *equivalence*, an important new concept implemented by the == binary message. *Objects x and y are equivalent if they are one and the same object*, in other words, if they are identical. Compare this with *equality*, a relation implemented with the = message: *Two objects are equal if their values are equal in some well-defined sense*. Obviously, two equivalent objects are equal (because they are the same), but the opposite may not be true as in the following example: When VisualWorks Smalltalk executes

```
| x y |  
x := 'John Smith'.  
y := 'John Smith'
```

the first assignment statement creates an internal representation of the string 'John Smith' and stores it in memory. The second statement again creates an internal representation of the string 'John Smith' and stores it in memory. We now have two different representations of 'John Smith' (Figure 4.18) which are equal in the sense of string equality (the corresponding characters of the two strings are the same) - but *not* equivalent because each has its own identity, its own bit pattern stored in its own place in memory.

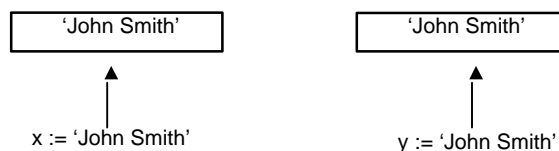


Figure 4.18. Two strings may be equal while not being equivalent. $x=y$ is true but $x==y$ is false.

On the other hand, x and y in

```
| x y |  
x := 'John Smith'.  
y := x    "y is bound to the same object as x."
```

are not only equal but also equivalent because both y and x are bound to the same object (Figure 4.19).

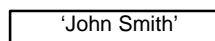




Figure 4.19. `y := x := 'John Smith'` binds `x` and `y` to the same object and both `x=y` and `x==y` are true.

As another example, consider that class `True` and `False` each have only one instance. As a consequence, equivalence and equality of `Boolean` objects mean the same thing.

The check of equivalence is faster because it only compares the memory addresses of the internal representation of the two objects. If the addresses are the same, the objects are equivalent; they are one and the same object. Checking whether two objects are equal may take much longer. As an example, to decide whether two non-equivalent strings are equal, we would probably first check whether they have the same size (if not, they cannot be equal) and if they do, we would have to check the individual character pairs one after another. Imagine how long this will take if the two strings are, for example, two versions of this book!

If the test for equivalence is so much simpler, why do we need equality? There are at least two reasons. One is that restricting equality to equivalence would sometimes be very inefficient. Consider again strings. If two strings that contain the same characters had to share one memory representation², then every time that we change a string, we would have to check whether we already have a string with the same characters. If we do, we must discard one and make all reference to it point to the already existing instance. This would be extremely inefficient.

Another reason why we need both equality and equivalence is that two objects are sometimes 'equal' even though they are not quite the same. As an example, two \$100 bills are interchangeable in terms of their value even though they are not one and the same object. In fact, a 'currency object' consisting of two \$50 bills and is also interchangeable with a \$100 currency object.

As another example, consider filenames. On some platforms, filenames are case insensitive, and strings such as 'filename' and 'FileName' are then two different versions of the name of the same file. Consequently,

```
'abc' asFilename = 'ABC' asFilename "true on platforms with case insensitive filenames."
```

On case insensitive platforms, we could thus define `=` for filenames as

```
= aFilename  
^ self string asLowercase = aFilenameString string asLowercase
```

where we assumed that the name of a filename is accessed by message `string`.

When should we use equivalence and when should we use equality? The answer is that you should *always use equality* unless you want to check that two references refer to the same object. In those situations in which equivalence and equality mean the same thing, they are defined to mean the same thing and there is no loss of speed in using equality. This definition is inherited from `Object` where

```
= anObject  
  
^self == anObject
```

It is very important to note that when `=` is redefined, the `hash` method must also be redefined. We will see what `hash` is and why it is so closely related to equality later.

² For special needs, there is a subclass of `String` called `Symbol` in which two identical values are stored only once and share the same location

Main lessons learned:

- Exclusive *or* of two Boolean arguments returns true if and only if the arguments are different.
- Two objects are equivalent (message `==`) if they are one and the same object. Equality (message `=`) means that two objects are in some sense interchangeable.
- The definitions of `=` and `==` inherited from `Object` give the same result for equality and equivalence. Many classes, however, redefine equality to satisfy their special meaning of 'interchangeable'.
- Testing for equivalence is very simple and fast, testing for equality may take much longer.
- Equivalence is stronger than equality: If two objects are equivalent, they are also equal, but two equal objects may not be equivalent.
- Always use equality unless you need to know that two expressions refer to the same object.

Exercises

1. Draw the truth table of exclusive *or*. Can *xor* be short-circuited? If not, how does this relate to the fact that the argument of *xor*: is a Boolean and not a `BlockClosure`?
2. List all combinations for which the ice cream - cake order test returns false.
3. Implement the following problems:
 - a. Female patients who have a sore throat and a headache, and male patients who have a sore throat but not a headache get medication M1; other patients don't get any medication. Write a code fragment to request gender and the medical problem and advise on the medication.
 - b. A qualified programmer will be hired if she wants to work in Santa Barbara and accepts a salary below \$130,000, or if she requires \$145,000 but accepts a location not in Santa Barbara. The program obtains the information and reports whether the candidate is acceptable.
4. Study the definition of *xor*: and comment on its consistency with the definition of *xor*. Are there any situations in which the definition will not produce the expected result?
5. Define equality of complex numbers with real and imaginary part. Note that it must be possible to compare complex numbers with other kinds of numbers.
6. Give your own example of a situation in which two objects should be considered equal even though all their components are not the same.
7. Count how many classes have their own definitions of `=` and explain two of them in detail.

4.9 Use of Booleans to repeat a block of statements

Almost all programs require repetition of a block of actions until some condition succeeds or fails. As an example, a search for a book in a library catalog must search catalog entries one after another until the desired entry is found or the search reaches the end of the catalog.

The process of repeating a block of actions until some condition is satisfied or fails is called *iteration* or *enumeration* and Smalltalk contains several methods to implement it. In the rest of this chapter, we will look at the most general ones and leave the more specialized for later.

Example 1: Print all positive integers whose factorial is not bigger than 15,765

Solution: The obvious solution is as follows:

1. Define a variable called, for example, `number` to hold the current value of the value over which we are iterating. Initialize `number` to 1.
2. Check whether `number factorial` is less or equal 15765. If yes, print the number, increment it by 1, and repeat this step, otherwise stop.

This algorithm can be implemented as follows:

```
| number |  
number := 1.
```

Transcript clear.

```
[number factorial <= 15765]      "This is the condition for doing another iteration"  
  whileTrue: [Transcript show: number printString; cr. number := number + 1]
```

In this example, iteration is performed with the whileTrue: message. Its general form is

conditionBlock whileTrue: iterationBlock

and it works as follows (Figure 4.20): The conditionBlock is evaluated; if the result is true, the iterationBlock is evaluated and conditionBlock is re-evaluated. Iteration continues until evaluation of conditionBlock returns false. Note that conditionBlock must always evaluate to true or false; if it does not, the message fails and opens an Exception Window.

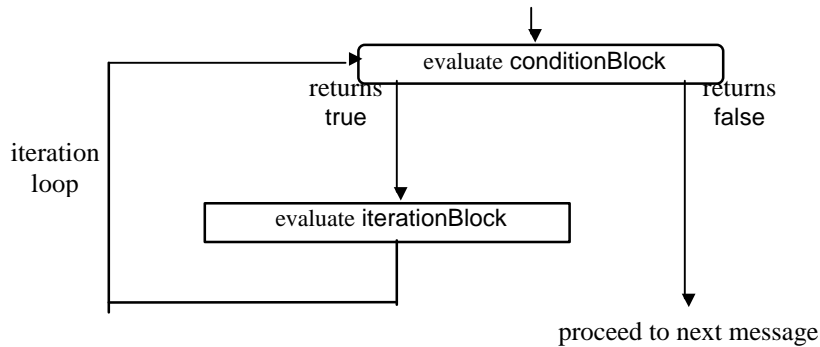


Figure 4.20. Evaluation of conditionBlock whileTrue: argumentBlock is based on looping.

The whileTrue: message is very useful but it must be used very cautiously. The danger is that if the receiver block never returns false, the program will continue looping around and never stop, a situation known as an *infinite loop*. Our program, for example, would get into an infinite loop if we forgot to increment the value of number in the iteration block.

In spite of all possible caution, you will undoubtedly eventually write a program that does get into an infinite loop and something that should take milliseconds will be running, and running, and running. If this happens, interrupt execution by pressing the <Ctrl> and <c> keys simultaneously. Smalltalk will open an Exception Window as if you halted the program with self halt and you can correct the program using the debugger or terminate execution.

Example 2: A simple shopping calculator

Problem: As a part of a Smalltalk application running a portable shopping calculator, we need a function that lets the user enter prices of grocery items, print them, and calculate and display a running total. When the total exceeds a preset limit, the program displays the total and displays a warning message.

Solution: The real application will require several classes and a graphical user interface (GUI) but since we are not in a state to implement it yet, we will test the general idea on a code fragment and a specific limit value, and display the required data in the transcript. The algorithm is as follows:

1. Initialize variable total to 0 and clear the Transcript.
2. While total <= 100 is true, do the following:
 - get price of the next item
 - display price in Transcript
 - update total
3. Display warning window and print total in Transcript.

This algorithm can be easily converted into the following program:

```
| price total |
```



```
"Initialize total and clear Transcript."  
total := 0.  
Transcript clear.  
"Keep gathering item prices and displaying them until the total exceeds 100."  
[total <= 100]  
  whileTrue:  
    [price := Dialog request: 'Enter next price' initialAnswer: "].  
    Transcript show: price; cr.  
    total := total + price asNumber].  
"Open warning window, print result, and stop."  
Dialog warn: 'The total has exceeded 100, end of input'.  
Transcript show: 'The total is now ', total printString
```

The whileFalse: message

Iteration is also available in the form whileFalse: which repeats execution as long as the receiver is false. You can always switch between whileTrue: and whileFalse: and the better choice is the one that makes more sense in the current context. With whileFalse:, the previous program could be rewritten as

```
| price total |  
"Initialize total and clear Transcript."  
total := 0.  
Transcript clear.  
"Gather item prices, display them, calculate running total."  
[total > 100] "Inverted condition."  
  whileFalse: "Instead of whileTrue:"  
    [price := Dialog request: 'Enter next price' initialAnswer: "].  
    Transcript show: price; cr.  
    total := total + price asNumber].  
"Open warning window, display result, and stop."  
Dialog warn: 'The total has exceeded 100, end of input'.  
Transcript show: [The total is now ', total printString, ' - more than 100']
```

The definition of whileTrue: is based on recursion

How is the whileTrue: method defined? Since the receiver is a block, its definition is in class BlockClosure and the definition is as follows:

```
whileTrue: aBlock  
  ^self value ifTrue: [aBlock value.  
                     [self value] whileTrue: [aBlock value]]
```

This is a bit obscure and requires an explanation. The first line of the definition

```
  ^self value
```

evaluates the receiver, just as we would expect from Figure 4.20, and sends it the ifTrue: message.

If the returned value is false, the ifTrue: message has no effect and execution is finished. If the returned value is true, execution continues with

```
  [aBlock value.  
  [self value] whileTrue: [aBlock value]]
```

Here aBlock value takes the argument block and evaluates it as in Figure 4.20, and execution proceeds to

```
  [self value] whileTrue: [aBlock value]
```

which is equivalent to

```
[self value] whileTrue: aBlock
```

Since `self value` is the evaluated value of the original block, `[self value]` is the same as the original block `self`. As a consequence,

```
[self value] whileTrue: aBlock
```

is the same as

```
self whileTrue: aBlock
```

This agrees with Figure 4.20 which shows that when we finish one iteration, we do another one, just like the one before. In effect, when the condition of `whileTrue:` succeeds (returns `true`), the `whileTrue:` message is sent again, and again, and again, until the condition finally fails.

A message that sends itself is called *recursive* and the concept is related to recursive definitions which we have already encountered.

Although recursion looks like magic, it is very natural even in real life situations. Consider, as an example, a person asking how to find the railway station in a large city. If the railway station is far, the best answer might be something like 'Go to the third intersection and ask again.' This advice is recursive because it requires re-execution of the original request.

Recursion often considerably simplifies the solution of a problem but it is often very inefficient because each recursion step produces a new message send and this results in extra execution time and memory overhead. For this reason, Smalltalk tries not evaluate `whileTrue:` using recursion and the compiler converts `whileTrue:` messages to in-line code whenever it can. The in-line code - direct translation into a sequence of machine instructions - will work faster and requires less memory.

The situations under which the definition of `whileTrue:` is in-lined is described in the comment of `whileTrue:` which goes essentially as follows:

"This method is in-lined if both the receiver and the argument are literal blocks. In all other cases, the code above is run.

Note that the code above is defined recursively. However, to save time and memory each time this method is invoked recursively, we have used the '[...] whileTrue: [..]' form in the last line, rather than the more concise 'self whileTrue: aBlock'. Using literal blocks for both the receiver and the argument allows the compiler to inline #whileTrue:, which could not be done if we were to use 'self whileTrue: aBlock'."

The comment refers to the concept of a *literal block* which is a block of statements surrounded by square brackets such as

```
[Transcript clear]
```

In other words, a literal block is a block that is directly recognized by the compiler. On the other hand, the block stored in the `aBlock` argument of the definition of `whileTrue:` is not literal because the compiler cannot determine whether the value of `aBlock` is a block or not. In fact, if we wrote something like

```
[x < 3] whileTrue: Transcript clear
```

the argument would not be a block but the expression should at least start executing because the receiver (a `ClockClosure`) understands the message.

The comment in the definition of `whileTrue:` means that if the argument of `whileTrue:` is a literal block, the compiler can recognize it as such and create in-line code. If the argument is not a literal block, the compiler cannot create in-line code because it does not have enough information. In this case, it will generate code to execute the definition (rather than creating in-line code as it would otherwise) by sending messages in the usual way. As an example, the `whileTrue:` message in

```
| block x |  
x := 1.
```

```
block := [x := x + 1].  
[x squared < 10] whileTrue: block.  
Transcript show: (x - 1) printString
```

will be executed by message sends (executing the whole definition of `whileTrue:`) rather than in-lined because the argument is not a literal block. When you attempt to execute this example, Smalltalk will open the window in Figure 4.21 which warns you that the argument of `whileTrue:` is normally a literal block and asks you if you really intend to use a non-literal. In this case, a non-literal is exactly what we want so click *proceed* and the program executes as expected.

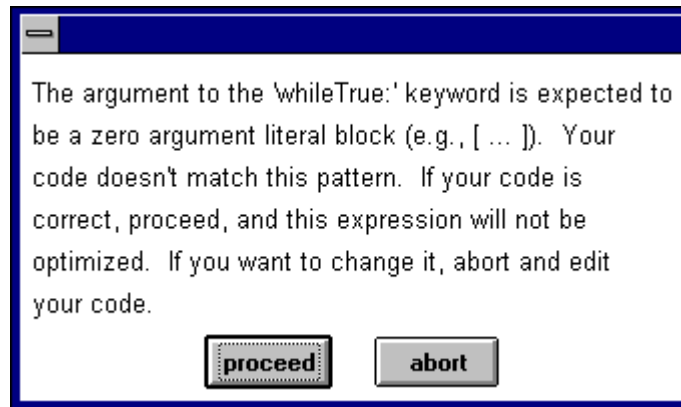


Figure 4.21. Warning produced when the argument of `whileTrue:` is not a literal block.

Besides the point that we have just explained, the comment has another even more obscure part that deals with the case when the argument is *not* a literal block. The essence is that if the compiler cannot in-line the original formulation, the recursive part is formulated so that the compiler *can* in-line the second execution and thus execute all iterations beyond the first one more efficiently. Consider the definition again:

```
whileTrue: aBlock  
  ^self value  
  ifTrue:  
    [aBlock value.  
     [self value] whileTrue: [aBlock value]]
```

If `self` or `aBlock` are not literal blocks, the message cannot be in-lined and the previously compiled definition executes. Now consider how this code has been compiled. Where the body contains a `whileTrue:` message:

```
[self value] whileTrue: [aBlock value]
```

the receiver and argument are both literal blocks and the compiled code is thus in-lined. Although the first iteration was thus executed less efficiently as a message, the remaining iterations are efficient because they use the in-lined code. Only the first iteration will thus be truly recursive, saving both execution time and memory space. This saving would not be possible if the formulation of the iteration was the more natural

```
self whileTrue: aBlock
```

because this code would again use a non-literal block.

Main lessons learned:

- Iteration is repeated execution of a block of statements. In its `whileTrue:` form, iteration stops when the receiver block returns `false`, in the `whileFalse:` form, it stops when the receiver block returns `true`.
- If the stopping condition is never satisfied, an infinite loop occurs. To interrupt an infinite loop, press `<Ctrl>` and `<c>` simultaneously.
- Recursion is the technique of defining a method in terms of itself.
- Recursion often leads to very simple solutions but it is usually inefficient in terms of speed and memory space.

Exercises

1. What is the value returned by the `whileTrue:` message?
2. Solve the following problems:
 - a. Open a confirmer asking the user whether to stop or not. If the answer is yes, the program stops, otherwise the confirmer is displayed again.
 - b. Print the square of all numbers between 10 and 20 in the Transcript.
 - c. Print all numbers that are between 1 and 1,000 and are divisible by 11 or 13 in the Transcript.
3. The factorial of a positive integer n is defined as follows: When $n=1$, the factorial of n is 1. When $n>1$, the factorial of n is n times the factorial of $n-1$. Write and test a recursive method derived directly from this definition. (The factorial method in the library is not defined recursively.)
4. The iteration methods covered in this section are all defined in class `BlockClosure`. Could they be redefined in the `Boolean` classes? If so, how?
5. What is the advantage of using a block rather than a `Boolean` expression as the receiver of a message?

4.10 Other forms of iteration

To close this chapter, we will introduce messages that strip the `whileTrue` and `whileFalse` ideas to a bare minimum by removing the argument. This means that all work done during the iteration must be done in the condition block. These messages have the following form:

<code>aBlock whileTrue</code>	"Repeats until <code>aBlock</code> evaluates to false."
<code>aBlock whileFalse</code>	"Repeats until <code>aBlock</code> evaluates to true."
<code>aBlock repeat</code>	"Repeats forever or until termination is forced inside <code>aBlock</code> ."

When you examine the library, you will find that it does not use these messages. (The references that you will find are not message sends.) Yet, they are useful and most of the uses of `whileTrue:` and `whileFalse:` can be easily converted into this form.

Example 1: Find and print the largest integer number x for which $x + (270 \cdot x^2) < 100,000$

This problem can be solved either with

```
| x |  
x := 1.  
[x + (270 * x squared) < 100000] whileTrue: [x := x+1].  
Transcript show: (x -1) printString
```

or with

```
| x |  
x := 0.  
[x := x + 1. x + (270 * x squared) < 100000] whileTrue.  
Transcript show: (x - 1) printString
```

The first solution is more natural and easier to read which is probably why `whileTrue` is not used. To implement the same task with `repeat`, we would have to change the structure to

```
| x |  
x := 0.  
[x := x + 1.  
 x + (270 * x squared) < 100000  
 ifTrue: [Transcript show: (x - 1) printString. ^self]] repeat
```

where we force exit from the loop with the return operator.

Example 2: Finding the root of a function

Problem: We need to find an approximate solution of

$$1 + \sin(x) = x$$

Solution: The problem can be solved by converting the equation into

$$f(x) = 1 + \sin(x) - x = 0$$

and finding the root of $f(x)$ - the value of x that produces $f(x) = 0$. Since this approach can be used to find the solution of any kind of equation with one unknown, we decide to restate the problem as follows:

Problem: Define a general purpose method to solve

$$f(x) = 0$$

Solution: Perhaps the simplest approach is as follows (Figure 4.22): Start with a suitable first guess for x and calculate $f(x)$. Increment x by a fixed increment and calculate $f(x)$ again. If the new $f(x)$ has the same sign as the old $f(x)$, increment x . Keep repeating these steps until the new and the old value of $f(x)$ have a different sign. At this point, decrease the size of the step (for example, divide it by 3), change its sign, and proceed as before (in the opposite direction, because the sign is different). Keep going back and forth until the value of $f(x)$ becomes small enough or until the number of repetitions reaches a preset maximum.

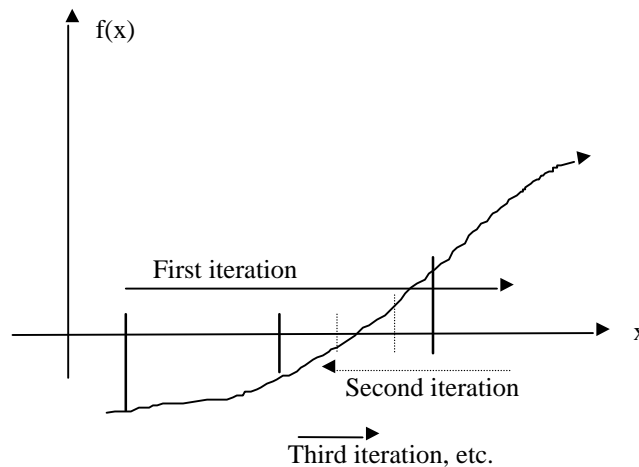


Figure 4.22. Finding root of $f(x) = 0$ by iteration.

We will now solve the problem in a simplified form by repeating the iteration until the two consecutive end-points are less than a predefined value δ apart. The algorithm is as follows:

1. Initialize starting value of x and use it as the current end-point.
2. Calculate the first value of $f(x)$.
3. While the termination condition fails, repeat the following steps:
 - a. While the old $f(x)$ and the new $f(x)$ have the same signs
Increment x and calculate new $f(x)$.
 - b. Divide increment by 3 and change its sign.
 - c. Re-evaluate the termination condition.
 - d. Store x as the current end-point.

Before writing the method, we will first test this algorithm on the following code fragment:

```
| delta x oldx step oldF newF condition |
"Initialize."
delta := 0.01.
step := 0.1.
x := 0.
oldx := 0.
condition := false. "Termination condition - initially not satisfied."
newF := 1 + x sin - x.
[condition] whileFalse:
    [[x := x + step.           "Increment x and calculate new value."
     oldF := newF.
     newF := 1 + x sin - x.
     oldF sign = newF sign] whileTrue.
    step := (step / 3) negated. "Update x and count."
    condition := (oldx - x) abs < delta.
    oldx := x].
"Output results."
Transcript clear; show: 'The approximate value of the root is x = ', x printString; cr;
show: 'The value of the approximation is f(x) = ', newF printString
```

When we executed this code, we obtained the following output:

The approximate value of the root is $x = 1.93457$
 The value of the approximation is $f(x) = -6.67572e-6$

This result is reasonable and we will thus convert the test fragment to a method.

The first question is what form the method should have. The method will need to know the function (this can be implemented as an evaluated block) and the following numbers: the starting point, the initial increment, and the value of delta. We thus decide on an implementation with selector `rootFrom:by:untilDelta:` that will be used as in

```
|function root |
function := [:x | 1 + x sin - x].
root := function rootFrom: 0.0 by: 0.1 untilDelta: 0.01.
Transcript clear;
    show: 'The approximate value of the root is x = ' , root printString;
    cr;
    show: 'The value of the approximation is f(x) = ' , (function value: root) printString
```

Note that we took the output out because the method could be used in situations in which no output is required.

The rest of this section is somewhat advanced for this point in the text and you may skip it without missing anything because we will return to the presented concepts later.

Where should the new method be defined? Our example shows that the argument is a block and so the definition will be in class `BlockClosure`. The implementation based on a few changes in our code fragment is as follows:

rootFrom: firstx by: firstStep untilDelta: delta

"Find root of self value = 0 starting from firstx in steps starting with firstStep. Use iteration back and forth until x end points differ by less than delta."

```
| x oldx step oldF newF condition |
"Initialize."
step := firstStep.
x := firstx.
oldx := 0.
condition := false. "Termination condition - initially not satisfied."
newF := self value: x.
[condition]
    whileFalse:
        [
            x := x + step. "Increment x and calculate new value."
            oldF := newF.
            newF := self value: x.
            oldF sign = newF sign] whileTrue.
        step := (step / 3) negated. "Update x and count."
        condition := (oldx - x) abs < delta.
        oldx := x].
^x
```

This definition requires several comments:

- We could not use arguments `firstx` and `firstStep` in the iterations because arguments cannot be assigned new values inside a method. We thus had to introduce temporary variables `x` and `step` and initialize them to `firstx` and `firstStep`.
- The block that represents `f(x)` must have an argument so that we can evaluate `f(x)` for a particular value. To be able to supply an argument into a block, we must use a block with an internal argument as in

```
[x | 1 + x sin - x]
```

Note the special syntax: `x` is inside the block, preceded by a colon and followed by a vertical bar. A block may have as many internal arguments as we need.

- To evaluate a block with an argument, we must send it an argument object and the value message is thus insufficient. For a block with one argument, use message `value:` instead as in

```
[ :x | 1 + x sin - x ] value: 0.4
```

When we tested the new definition with

```
[function root |  
function := [:x | 1 + x sin - x].  
root := function rootFrom: 0.0 by: 0.1 untilDelta: 0.01.  
Transcript clear;  
    show: 'The approximate value of the root is x = ', root printString;  
    cr;  
    show: 'The value of the approximation is f(x) = ', (function value: root) printString
```

we obtained the same result as before. The method is thus a correct translation of the original code fragment.

Main lessons learned:

- Methods `whileTrue`, `whileFalse`, and `repeat` don't have an argument block and all work must be done in the condition block..
- The use of `repeat` requires forced exit from the block.

Exercises

1. Solve the following problems from the previous section using `whileTrue`, `whileFalse`, and `repeat`.
 - a. Open a confirmer asking the user whether to stop execution or not. If the answer is yes, the program stops, otherwise the confirmer is displayed again.
 - b. Print the square of all numbers between 10 and 20 to the Transcript.
 - c. Print all numbers between 1 and 1,000 divisible by 11 or 13 to the Transcript.
2. Draw a flowchart describing the operation of the `whileFalse` method.

Conclusion

This chapter focused on Boolean objects and blocks. Booleans are used to represent the outcome of tests and yes/no questions. They are an essential component of most computer programs because they are the basis of decision making (should we execute a block of statements or not?), selection (should we execute block A or block B?), and iteration (should we execute the block again?).

In Smalltalk, Boolean objects are implemented by the abstract class `Boolean` which leaves the implementation of most of its methods to its subclasses `True` and `False`. Classes `True` and `False` each allow only one instance represented by literals `true` and `false`.

Another essential component of decisions, selections, and iterations is the block object, an instance of `BlockClosure`, a sequence of statements surrounded by square brackets. Class `BlockClosure` defines various forms of the `value` message which evaluate the statements inside the block in its proper context. Until a block object receives the `value` message, it is just a container holding a deferred sequence of statements.

Messages implementing decisions are defined in classes `True` and `False`. They include the one-keyword messages `ifTrue:` and `ifFalse:` and the two-key word messages `ifTrue:ifFalse:` and `ifFalse: ifTrue:`. All use blocks as arguments. Classes `True` and `False` also implement logic operations *and*, *or*, *exclusive or*, and *not*. Operations *and* and *or* are implemented in two forms - fully-evaluating and non-evaluating. The non-evaluating message does not evaluate its argument unless it is required to obtain the Boolean result. It is thus faster and preferable. Besides, it can be used as a safeguard protecting illegal evaluation of the argument.

Iteration methods are defined in class `BlockClosure` and their receiver block must evaluate to true or false. All forms of iteration defined in `BlockClosure` except for `repeat` require the evaluation of a condition block which determines whether to iterate again or proceed to the next message.

The body of the definition of `whileTrue:` re-sends the `whileTrue:` message, a technique known as recursion. Recursive solutions are often simpler than iterative ones but they are usually less efficient because of the extra work and memory overhead required by each new message send. In most cases, recursion can be easily replaced by more efficient iteration.

Although the Smalltalk library contains formal definitions of `ifTrue:`, `ifFalse:`, and `whileTrue:`, the compiler usually does not use them when compiling Smalltalk code. Instead, it uses pre-defined machine code that implements the operation more efficiently. This technique is called in-lining and its use in Smalltalk is rather rare. The vast majority of method definitions in the Smalltalk library are compiled into message sends reflecting directly the original code.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

BlockClosure, **Boolean**, *Date*, *Dialog*, **False**, *Time*, **True**.

Terms introduced in this chapter

and - Boolean operation; returns true if and only if the receiver and the argument are both true

Boolean - general term denoting true and false objects and operations on them; basis of tests

block - instance of class `BlockClosure`; sequence of statements surrounded by square brackets and evaluated together; to evaluate it, send it message `value`

body - last part of method definition - sequence of statements implementing method behavior

confirmer - a dialog window offering a yes/no choice

conjunction - another name for the *and* operation

conversion message - message converting an object into a related object

dialog - window-based interactive communication between the user and a program

disjunction - another name for the *or* operation

equality - two objects are equal if their values are in some sense interchangeable; the `=` message

equivalence - two objects are equivalent if they are identical, one and the same object; the `==` message

exclusive or - Boolean operation; returns true if and only if the receiver and the argument are different
Booleans

file-in - reading and compiling a file-out file and inserting its contents into the library

file-out - storing a method, protocol, class, or category in a file using a special format suitable for a file-in

flowchart - a diagram showing the order of execution of an algorithm

fully-evaluating and/or - implementation of *and/or* in which both the receiver and the argument are always evaluated

heading - the first part of method definition ; specifies the selector and argument names, if any

in-line code - machine code inserted by the compiler into method translation instead of code to invoke a message; used for a few frequently used messages to increase speed of execution

iteration - repetition of a block of statements until some condition is satisfied

literal - an object created by the compiler from its textual form without a creation message

logic - synonym of **Boolean**

method definition - formal specification of the implementation of a message

non-evaluating and/or - implementation of *and/or* that evaluates the argument only if necessary

notifier - a dialog window displaying text and offering only an *OK* button for continuation; created with the `warn:` message to class `Dialog`

or - Boolean operation; returns false if and only if both the receiver and the argument are false

recursive - referring to itself

singleton - the single instance of a class that does not allow multiple instances

truth table - a table defining a Boolean operation by listing all combinations of *true* and *false* operand values and the corresponding value of the result

Chapter 5 - Numbers

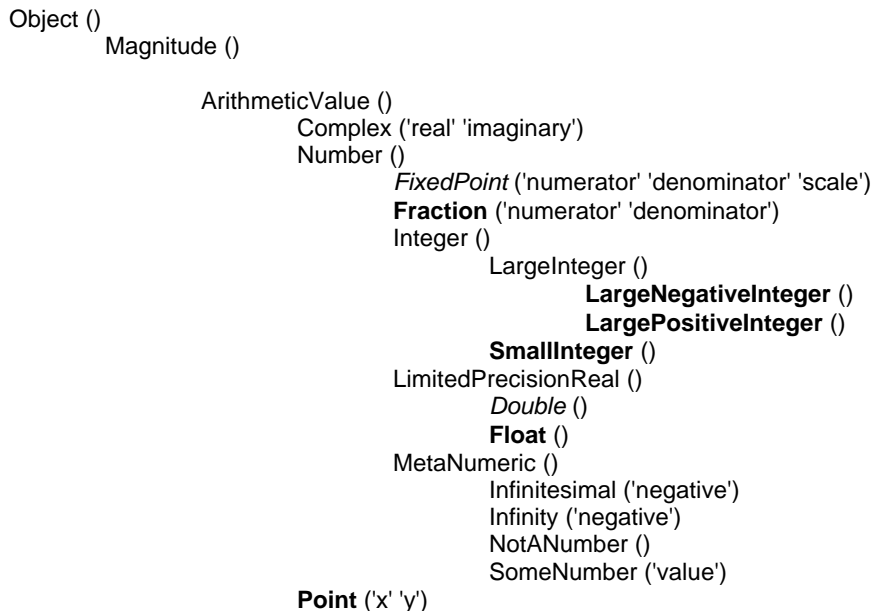
Overview

VisualWorks library contains many classes representing numbers of various kinds, mainly because computers internally represent and process different kinds of numbers differently. In addition, computer programs often require facilities that go beyond basic hardware capabilities and Smalltalk number classes satisfy these additional needs.

Although the main use of numbers is for calculations, numbers are also used to implement iteration. This includes execution of a block of statements a fixed number of times or over a range of numbers with given start and end values. Several methods implementing these very important functions are included in the library.

5.1. Numbers

All Smalltalk number classes are defined in the Magnitude - ArithmeticValue part of the class hierarchy. Its structure is as follows:



In this diagram, the most important classes are **boldfaced**, less important classes are *italicized*, and classes rarely encountered are in regular font.

The hierarchy starts with the abstract class Magnitude which gathers the general properties based on comparison¹. Subclasses of Magnitude include mainly number classes but also other classes whose instances can be compared such as Date, Time, and Character. Numeric subclasses of Magnitude are collected under ArithmeticValue, another abstract class, which gathers methods shared by all numbers such as addition, subtraction, and absolute value. Concrete definitions of most of these methods are left to concrete subclasses but some are defined here, for example changing the sign of an instance by methods abs and negated.

Some of the subclasses of ArithmeticValue, such as Complex, are little used and Complex and MetaNumeric are not even included in the standard library and must be filed in. All the frequently used kinds of numbers are subclasses of the abstract class Number which is responsible for scalar entities -

¹ Note that although most classes that understand comparison are gathered under Magnitude, some important classes such as strings are located under a different class.

numbers that have magnitude but cannot be used as vectors - indicators of direction or coordinates of points in space. (Instances of `Complex` and `Point` are related to vectors.) `Number` defines methods such as division and mathematical functions such as `sin` and `cos`, mostly on the basis of conversion to floating-point and delegation to floating-point number classes. A large conversion protocol is included which allows conversion of almost any kind of number to almost any other kind.

The concrete subclasses of `Number` are `LargeNegativeInteger`, `LargePositiveInteger`, and `SmallInteger` (all of them numbers without a decimal point), `Fraction` (based on integer instance variables `numerator` and `denominator`), `FixedPoint` (with instance variables `numerator`, `denominator`, and `scale`), `Double` and `Float` (numbers with decimal point and fractional part), and special classes `Infinitesimal`, `Infinity`, `NotANumber`, and `SomeNumber` - all subclasses of `MetaNumeric`. If you are interested in numbers, the implementation of some of the methods provides rich material for the study of very interesting algorithms.

Before introducing the most important properties of number classes in detail, we will now briefly outline their purpose and functionality.

Integers

Integers are numbers without a decimal point such as 13, -45700089, and 0. Computers implement integers and integer arithmetic differently from other kinds of numbers, and the number of bits that they use to store integers determines the maximum size that they can handle in hardware. In VisualWorks Smalltalk, integers directly processed by computer hardware are implemented by class `SmallInteger` and their maximum and minimum values can be obtained as follows:

SmallInteger minVal	"Returns, for example, -536870912"
SmallInteger maxVal	"Returns, for example, 536870911"

Although the exact value of the limit depends on your computer's CPU, it is obvious that the range of `SmallInteger` is not so small after all. In fact, the name `SmallInteger` is due to the fact that `Smalltalk` also implements essentially unlimited integers using two 'large integer' classes. One special feature of `SmallInteger` objects is that they are among the few objects that are not accessed by pointers: Whereas the most other objects are represented by the memory address of their detailed description (a pointer to the internal representation), `SmallInteger` objects are directly represented by the binary code of the integer value itself; this makes their access very efficient.

The idea behind introducing special classes for large integers is that if your computer has a lot of memory, there is no reason why the available bytes could not be used to store a representation of very large magnitudes, in principle an unlimited sequence of digits. Unlike `SmallInteger` objects, such numbers cannot be processed by a single CPU instruction and require special treatment which is implemented by classes `LargeNegativeInteger` and `LargePositiveInteger`, both subclasses of `LargeInteger`. These ‘large integers’ allow arithmetic without any concern about magnitudes. As an example, `LargePositiveInteger` allows you to calculate the factorial of 200, in other words, the number obtained by multiplying $200 \cdot 199 \cdot 198 \cdot 197 \dots \cdot 3 \cdot 2 \cdot 1$. The expression

200 factorial

returns

7886578673647905035523632139321850622951359776871732632947425332443594499634033429203
0428401198462390417721213891963883025764279024263710506192662495282993111346285727076
3317237396988943922445621451664240254033291864131227428294853277524242407573903240321
2574055795686602260319041703240623517008587961789222227896237038973747200000000000000
00

This is a result that few other languages would let you obtain as easily. Although there are not many applications that need to calculate the factorial of 200, unlimited numbers are very useful in business and other applications. (Unfortunately, national debts easily go beyond the limits of `SmallInteger`.) There is, of course, a price for this computational power and this price is that calculations with `LargeNumber` take

considerably longer than calculations with `SmallInteger`. We will see how much longer later, when we show how to measure the time that it takes to execute a statement.

The unlimited range of Smalltalk numbers is a valuable feature but if you had to treat ‘small’ and ‘large’ integers differently and keep track of the boundaries between them during arithmetic, the advantage of having unlimited integers would be small. This is why Smalltalk performs conversion between the three classes automatically and you don’t even have to know that the three classes exist; all you need to understand is the concept of an integer. Technically speaking, implementation of integers is *transparent* to the programmer.

Floating-point numbers

Floating-point numbers are numbers with a decimal point such as 3.14, -789000000.123, 0.000000341, 0.0 or 1.33185e17 (1.33185 * 10¹⁷). Like most other languages, Smalltalk has two floating-point representations implemented by two floating-point classes - Float ('single precision') and Double ('double precision'). The difference between them is how many bytes they use and how much precision and what range they provide. Unlike the automatic conversion between SmallInteger and LargeInteger, conversion between Float and Double is not automatic and requires conversion messages asFloat and asDouble. The comment of Double explains the principle of the class as follows:

Instances of class `Double` represent floating-point numbers in IEEE 64-bit format.

These floating-point numbers are good for *about 14 or 15 digits of accuracy*, and the range is between plus and minus 10^{307} .

Here are some valid floating-point examples:

8.0 13.3 0.3 2.5d6 1.27d-30 1.27d-31 -12.987654d12

The format consists mainly of no imbedded blanks, little d for tens power, and a *digit on both sides of the decimal point*.

The bizarre phrase ‘*about 14 or 15 digits of accuracy*’ reflects the fact that internal representation of floating-point numbers is binary and conversion of decimal values to binary usually results in loss of accuracy.

Representation using letter d is mainly for very large and very small magnitudes and the letter separates the magnitude part from the exponent part. As an example

$$1.27d-30 = 1.27 * 10^{-30} = 0.000000000000000000000000000000127$$
$$-12.987654d12 = -12.987654 * 10^{12} = 12987654000000.0$$
$$127d3 = 1.27 * 10^5 = 127000.0$$

Note that .123d is illegal because the decimal point must be surrounded by digits.

When you *inspect* a Double number, such as the Double class constant Pi (returned by class message Double pi) the inspector shows that Double numbers use eight bytes which agrees with the comment which states that 64 bits are used. The actual bytes are normally of no interest to the programmer.

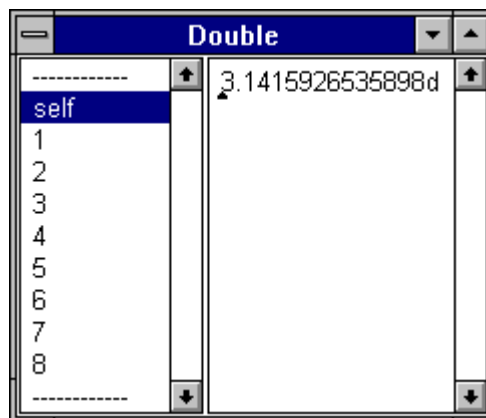


Figure 5.1. Inspector on Double class constant Pi obtained by message Double pi.

The comment for class Float is similar to the comment for Double, the main difference being the range and accuracy. Note also that Float representation uses letter e instead of letter d for the exponent part of the code:

Instances of the class Float represent floating-point numbers in platform-dependent short float format. These floating-point numbers are good for *about 8 or 9 digits of accuracy*, and the *range is between plus and minus 10³⁸*.

Here are some valid floating-point examples:

8.0 13.3 0.3 2.5e6 1.27e-30 1.27e-31 -12.987654e12

The format consists mainly of no imbedded blanks, little e for tens power, and a *digit on both sides of the decimal point*.

An important point about floating-point numbers is that the displayed value is not exactly what is stored in memory because of the incompatibility of the internal binary representation and the decimal representation on the screen. The number of digits displayed on the screen is controlled by the value returned by the message defaultNumberOfDigits:

Float defaultNumberOfDigits

returns 6, and

Double defaultNumberOfDigits

returns 14 which explains the Inspector in Figure 5.1. Both Float and Double numbers are thus displayed with less than their full accuracy. You can change this constant if you wish by changing the definition of defaultNumberOfDigits but no matter how many digits you print, the internal representation is fixed and changing defaultNumberOfDigits will only change the display but not the precision of arithmetic.

It is useful to note that the library contains a class that converts numbers to formats other than the default. This class is called PrintConverter and the display method is print: number formattedBy: string. It is used as follows:

PrintConverter print: 1000 * Float pi formattedBy: '####,###.##' "Returns 3,141.59."

but other formats are also available.

Fractions

Instances of class Fraction represent values such as -3/5 or 14/27. Each fraction is internally represented by two integer objects - an integer numerator and an integer denominator - so that there is no conversion and no loss of accuracy in arithmetic. Arithmetic on Fractions uses the well-known principles of combining numerators and denominators. As an example, the Fraction method for addition is defined essentially as follows²:

+ aFraction

^Fraction numerator: numerator * aFraction denominator + (aFraction numerator * denominator)
denominator: denominator * aFraction denominator

The body first calculates the value of the numerator and denominator, and then creates a new fraction with these numerator and denominator values. The process requires the creation of a new instance, three multiplications (notoriously time-consuming), and one addition; it is thus clear that addition of Fraction objects will take much longer than addition of Integer objects. The redeeming advantage of

² In reality, the definition uses double dispatching which is explained later in this section.

fractional arithmetic is that it fully preserves accuracy, which is not true for floating-point representation which is inherently inaccurate.

Fixed point

According to the comment of class FixedPoint

FixedPoint numbers represent 'business' numbers - numbers with arbitrary precision before the decimal point, but limited precision after the decimal point. A prime example of their use is in expressing currency values, which are always rounded off to the nearest hundredth, but which could easily have more digits than a Float (or possibly even a Double) could accurately express.

Main lessons learned:

- Smalltalk's rich hierarchy of number classes is rooted in the abstract classes Magnitude, ArithmeticValue, and Number.
- The most important types of numbers are integers, floating-point numbers, and fractions. Each uses a different internal representation and performs arithmetic differently. Some types of numbers are directly implemented in hardware, others are not, and this results in different ranges, accuracy, speed of arithmetic, and memory requirements.
- Smalltalk integers consist of 'small' integers (directly implemented in hardware) and 'large' integers for essentially unlimited magnitudes. Smalltalk automatically converts between the different forms when this is required by the calculation.
- Floating-point numbers represent values with a decimal point. They are implemented by classes Float and Double and the difference between them is in accuracy and range. Like small integers, floating-point arithmetic is directly implemented by hardware.
- Floating-point numbers are inaccurate binary representations of their decimal counterparts.
- Class Fraction provides uncompromised accuracy based on integers at the cost of very slow execution.
- Class FixedPoint is an alternative of Fraction intended for decimal data.

Exercises

1. List all concrete method definitions in ArithmeticValue and Number.
2. Methods fully defined in abstract classes usually depend on methods defined in concrete subclasses. These methods, left as *subclass responsibility*, should be listed in the class comment. Read the comment of Magnitude, list methods that must be defined in subclasses, and methods that this "buys you", in other words, methods that need only be defined in Magnitude and will work for any or almost any subclass.
3. Write a half page summary of class MetaNumeric.
4. Write a half page summary of class Complex.
5. If you are familiar with the principles of floating-point representation, examine how class Float represents floating-point numbers.
6. How many digits does 2000 factorial have? (Hint: Message size returns the number of characters in a String.)
7. To gain insight into floating-point numbers, step through the execution of the following code fragment:
self halt.
Transcript show: 3.1 printString
8. Write a short description of the most useful features of class PrintConverter.
 9. Class Magnitude gathers shared behavior of all objects that can be compared. Most of its subclasses are numbers but another very important subclass is Character which is mainly used for the representation, conversion, and testing of characters including letters, digits, and punctuation symbols. Read the comment of Character and list the most important methods in the *converting*, *testing*, and *accessing untypeable characters* protocols.
 10. Characters are a subclass of Magnitude because they can be compared. List all useful methods that they inherit.

11. Class `ArithmeticValue` disables the `new` message. This means that numbers cannot be created with `new` - except for class `Complex`. Examine and explain how class `Complex` gets around this limitation.

5.2 Operations on numbers

Most number operations can be performed on all types of numbers with the usual limitations such as no division by 0, logarithms and square roots of positive numbers only, and so on. We will not list all the available messages because few applications require more than basic arithmetic. Besides, number messages are quite self-explanatory and don't require any comments; as an example, if you know what `arcsin` is, you will not have any problem finding it in the browser and using it.

We must regretfully admit that Smalltalk is not ideal for large amounts of mathematical calculations (especially on floating-point numbers) because its very principle – treating everything as objects - requires overhead that slows arithmetic down. If you want to write a program to calculate the state of the Universe in the first 30 nanoseconds after the Big Bang, don't use Smalltalk beyond the first prototype or implement the extensive calculations in another language. You may be better off using C or, better still, assembly language. For extensive floating-point calculations, benchmark measurements give a wide range of comparisons but C is probably about five times faster on the average than unaided Smalltalk.

After this introduction we will now examine the most common arithmetic, mathematical, and conversion messages.

Arithmetic messages

All numbers support all arithmetic operations including addition, subtraction, multiplication, and division.

<code>+</code>	work in the usual way on all combinations of numbers
<code>-</code>	
<code>*</code>	
<code>/</code>	produces a <i>Fraction</i> when both the receiver and the argument are integers, <i>Float</i> if floating-point numbers are involved
<code>quo:</code>	calculates the integer quotient of division with truncation towards zero
<code>rem:</code>	calculates the remainder of division in terms of <code>quo:</code>

Examples with integers:

<code>3 / 5</code>	"Returns fraction 3 / 5."
<code>17 quo: 3</code>	"Returns 5."
<code>17 rem: 3</code>	"Returns 2 because $5*3 + 2 = 17$."
<code>-17 quo: 3</code>	"Returns -5."
<code>-17 rem: 3</code>	"Returns -2 because $-5*3 + (-2) = -17$."

Examples with floating point numbers:

<code>3 / 5</code>	"Returns 0.6."
<code>1.7 quo: 0.3</code>	"Returns 5."
<code>1.7 rem: 0.3</code>	"Returns 0.2 because $5*0.3 + 0.2 = 1.7$."
<code>-1.7 quo: 0.3</code>	"Returns -5."
<code>-1.7 rem: 0.3</code>	"Returns -0.2 because $-5*0.3 + (-0.2) = -1.7$."
<code>rounded</code>	"Returns the integer nearest the receiver."
<code>truncated</code>	"Returns the integer nearest the receiver toward zero."
<code>1.7 truncated</code>	"Returns 1 - receiver stripped of its decimal part."
<code>1.4 truncated</code>	"Returns 1."
<code>1.7 rounded</code>	"Returns 2 - the nearest integer - compare with truncation."
<code>1.4 rounded</code>	"Returns 1."
<code>1.5 rounded</code>	"Returns 2."
<code>-1.7 truncated</code>	"Returns -1."
<code>-1.4 truncated</code>	"Returns -1."
<code>-1.5 rounded</code>	"Returns -2."
<code>13 even</code>	"Returns false - number 13 is not even."

Mathematical messages

Float pi/2 sin	"Returns 3.45497 - not $\sin(\pi/2) = 1$ as we expected. Why?"
(Float pi/2) sin	"Returns 1.0 - this is what we had in mind."
(Float pi/2) cos	"Returns -4.37114e-8 or -0.000000043711. Close to 0 but not exact."
(Float pi/4) tan	"Returns 1.0"
10 log	"Returns 1.0 because $10^1 = 10$ "
10 raisedTo: 0.5	"Returns 3.16228, the square root of 10"
10 sqrt	"Returns 3.16228 - same as the previous message, as expected."

Conversion messages

3.14 asInteger	"Returns 3, same as message truncated."
17 asFloat	"Returns 17.0"
Float pi asRational	"Returns (918253/292289) - approximates π as a fraction."
Double pi asRational	"Returns (130161051938194/41431549628009)"
180 degreesToRadians	"Returns 3.14159 because 180 degrees is the same as π ."

Other

2 max: 7	"Returns 7, the larger of the two numbers"
2 min: 7	"Returns 2, the smaller of the two numbers"

Example 1: Accuracy of conversion between floating-point numbers and fractions

Method asRational converts numbers to instances of Fraction. The message is understood by almost all numbers and defined in several classes but the only interesting definition is in class LimitedPrecisionReal; this definition is inherited by Float and Double. Its principle is conversion to continued fractions, numerical objects of the following kind:

$$1 + \frac{375}{1 + \frac{43}{1 + \frac{13}{1 + \dots}}}$$

To check how accurate the conversion is, we converted a floating-point number to a Fraction and then back to a floating-point number, and compared this with the original number as follows:

12345678.1476 asRational asFloat - 12345678.1476	"Returns 0.0"
12345678.76 asRational asFloat = 12345678.76	"Returns true."
31.76 asRational asFloat - 31.76	"Returns 1.90735e-6"
31.76 asRational asFloat = 31.76	"Returns false."

The calculation appears to be very accurate but we will make a more thorough test later.

Example 2: Defining a new arithmetic method

Problem: Incrementing a number by 1 is so common that it might be useful to have a special method to implement it as a unary message. The method could be used as in

```
| n |
n := 15.
some calculations here
n increment
```

Solution: The problem is simple and the solution is to take the receiver, add 1 to it, and return the result as in

increment

"Increment receiver by 1."
^self + 1

The method should be useful for any number and we will thus define it in class `Number`. Executing

15 increment

returns 16 and the method thus appears to work. However, executing

```
| n |  
n := 15.  
n increment.  
n
```

with *print it* returns 15 as the value of `n` and this does not appear to be correct. What is happening? The receiver of

`n increment`

is the object bound to `n`, in other words 15. The method takes the receiver, adds 1 to it, and returns 16. However, the binding of `n` to 15 is not affected and the value of `n` thus remains 15. Our fundamental goal thus has not been achieved but we learned an important lesson.

Main lessons learned:

- VisualWorks library includes many number classes, mainly because different kinds of numbers require different internal representation and different hardware implementation.
- Smalltalk is not ideal for problems involving large amounts of arithmetic, particularly when it involves floating-point numbers.
- Classes such as `LargeInteger` and `Fraction` provide extra range and accuracy found in few other languages. The cost is long processing time.

Exercises

1. Test the following messages in short examples:
 - a. `asFloat` (in particular in class `Fraction`)
 - b. `asRational`
 - c. `fractionPart`
 - d. `rounded`
 - e. `truncated`
 - f. `degreesToRadians`
 - g. `radiansToDegrees`
 - h. `squared`
2. Compare the results of `Float pi asRational` and `Double pi asRational`.
3. The result of division of integers, is automatically reduced. Test this by executing `4/6` with *inspect*.

5.3 Implementation of binary arithmetic - double dispatching and primitives

Smalltalk arithmetic is based on double dispatching. We will now explain this important principle and refer you to an alternative description in method `aboutDoubleDispatching` in the class protocol documentation in class `ArithmeticValue`.

Assume that you want to execute the message `3.14 + (5/8)`. The two operands are different kinds of numbers must first be converted to the same class. Since `Fraction` arithmetic is based on integers and since `3.14` cannot be converted to an integer without loss of accuracy, the `Fraction` is converted to `Float` and the two numbers can then be added using floating-point arithmetic.

In general, `3.14 + x` with an arbitrary argument could be implemented as follows:

1. Check if the argument is a `Float`. If it is, do `Float` addition and exit.
2. Check if the argument is a `Double`. If it is, convert the receiver to `Double`, do `Double` arithmetic and exit.
3. Check if the argument is a `SmallInteger`. If it is, convert it to `Float`, do `Float` arithmetic and exit.
4. Check if the argument is a `Fraction`. If it is, convert it to `Float`, do `Float` arithmetic and exit.
5. Check if the number is a `LargePositiveInteger`. If it is, try convert it to `Float`, do `Float` arithmetic, and exit.
6. And so on and on, a long chain of tests, conversions, and operations.

If the class of the argument is somewhere at the end of this chain, it will take a long time before we even know what to do. For larger amount of arithmetic, this would be a serious drawback because each test adds overhead. The code is also too complicated. And if we add a new number class, we will have to change the definition of all arithmetic messages in all existing number classes to deal with this new type of argument. All in all, this is an unattractive approach.

We had a similar problem before when we considered drawing different kinds of geometric objects and we solved it using polymorphism. To eliminate tests, we relied on the receiver to do 'the right thing'. Unfortunately, our problem is different - the reason for testing is the argument, not the receiver. The problem is thus 'turned around the operation' and we cannot use polymorphism. But maybe the solution can be based on turning the problem around - exchanging the receiver and the argument - and this is exactly what double dispatching does.

Smalltalk implementation of `Float` addition method first checks whether it can do addition using CPU code. If it can (which happens if the argument belongs to the same or a closely related class), it does, and execution of the method is finished. If it cannot, the code *exchanges* the receiver and the argument (in our example `3.14 + (5/8)`, the receiver is now `Fraction 5/8`) and sends a message to it (a `Fraction` object), indicating that the argument is a `Float`. The definition of the method in `Fraction` takes this hint, converts the receiver `5/8` to a `Float` (without any testing) and sends the `+` message (with the fraction converted to float) to the `Float` receiver. `Float` now executes `+` directly because both numbers are now `Float` objects (Figure 5.2). Message `+` is thus dispatched once to the original receiver and then, if necessary, again - to the argument. The result is that the arithmetic operation requires at most two extra message sends but no explicit checks of

the class of the argument. Since message sends require only a minimal number of CPU cycles, this approach is more efficient - and the code much neater. And if we add a new class, we only have to add new double dispatching messages and none of the existing messages is affected.

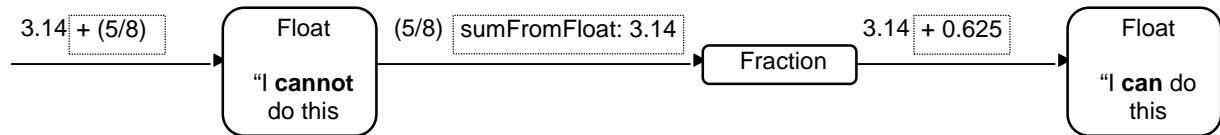


Figure 5.2. Execution of `3.14 + (5/8)` uses double dispatching.

We can now examine the definition of `+` in class `Float` which is as follows:

+ aNumber

"Answer a Float that is the result of adding the receiver to the argument.
The primitive fails if it cannot coerce the argument to a Float."

```
<primitive: 41>  
^aNumber sumFromFloat: self
```

The code means that if `<primitive: 41>` fails to convert the two numbers to a common form and add them (ignore the concept of a primitive for a moment) the *argument* `aNumber` becomes the receiver of the message `sumFromFloat:`. In our example, `aNumber` is a `Fraction`, the message thus goes to class `Fraction`, and the definition of `sumFromFloat:` in class `Fraction` is

sumFromFloat: aFloat

```
^aFloat + self asFloat
```

In our case, `self` is `5/8`, `self asFloat` is `0.625`, and the method thus sends `+ 0.625` to `Float`. This is the same `+` method as above but this time, the argument is a `Float` and the primitive executes and returns the sum. The operation is finally completed. We suggest that you test how this works using the Debugger on

```
self halt.  
3.14 + (5/8)
```

The definition above introduced the concept of a *primitive*, an expression written in a special syntax that does not indicate any message sends. A primitive is not implemented in Smalltalk but in a lower level language such as C or assembly language. In our example, `primitive 41` executes a machine language program that adds two floating-point numbers if both are floating-point numbers or if the program knows how to convert the second number to float. There are many other primitives, such as `primitive 1` which adds two small integers.

When a method contains a primitive, the primitive is always the first statement. If it fails to provide the desired result, execution continues to the next statement in the definition. The best way to think of a primitive is as a machine level instruction of some fictitious CPU. As an example, think of `primitive 1` used in the definition of `+` in `SmallInteger` as 'machine instruction number 1' which checks whether the argument is a `SmallInteger`, performs addition if it is or if it can be easily converted, and exits with some special result if it is not.

You will now probably say: 'Aha, after all, Smalltalk is not all that pure. In the end, it must descend to the level of the CPU and work just like any other language'. And you are quite right! Every programming language must, in the end, be translated to machine level and the binary codes must be executed by the CPU. But this is true for every language.

One last question concerning primitives remains: Where is the code that executes primitives? The answer is that the implementation of primitives is included in the Smalltalk *Virtual Machine* (VM), a

program that provides an interface between Smalltalk and the CPU. The VM is thus essential for Smalltalk execution and an application written in Smalltalk requires two parts to run: The Virtual Machine program (called, for example, `oe.exe` - derived from Object Engine, another name for VM) and the image file containing data related mainly to the class library and stored in a `.im` file. Creating a stand-alone Smalltalk application thus requires creating an image file containing information about the application's classes, and a copy of the VM file. If you are developing the library, you also need a changes file `.ch` which contains all the changes that you made to the library, and the `.sou` file which contains the source code in textual form. For proper operation, the `.im`, `.ch`, and `.sou` files must be synchronized which automatically happens if you use the versions saved by the `save` or `save then exit` commands.

The concept of a Virtual Machine has recently attracted general attention with the emergence of the Java language which is also based on a virtual machine. To make Java portable (executable on different CPUs), Java programs are translated into *bytecodes* which are identical across all CPUs, and executed by virtual machines which are tailored to specific CPUs. Smalltalk uses the same principle.

Main lessons learned:

- Smalltalk arithmetic uses the concepts of double dispatching and primitives.
- Double dispatching means re-sending (re-dispatching) a one-argument message to the argument, using the original receiver as the argument.
- Double dispatching eliminates tests because the dispatched message implies the type of the argument.
- The main use of double dispatching is in arithmetic but the principle has general validity.
- A primitive is an operation implemented at 'low level', possibly directly in machine code. Each primitive has a number and can be thought of as a machine code of a fictitious CPU.
- Primitive functions use special syntax of the form `<primitive: 1>`. If a method contains a primitive, the primitive is always the first statement; the remaining code is executed only if the primitive fails.
- Primitives are implemented via a virtual machine.
- A virtual machine (also called object engine) is a program that interfaces between Smalltalk and the CPU.
- The concept of a virtual machine makes it possible to compile code into one form shared across all platforms. The elements of this representation are called bytecodes. They can be executed on any machine that has a virtual machine for processing them.
- A completed standalone Smalltalk application requires a virtual machine and an image file.

Exercises

1. Find and explain the `+` and `/` methods in the following classes:
 - a. `SmallInteger`
 - b. `Double`
 - c. `Fraction`
2. Extend the principle of double dispatching from one argument to two arguments.

5.4 Using numbers for iteration - 'repeat n times'

In some problems, we need to evaluate a block of statements a known number of times. For example, we might want to flash a message on the screen three times, or execute a block 1,000 times to get an accurate measure of its execution time. Although all forms of repetition can be implemented with `whileTrue:`, iteration based on counting is so common that Smalltalk provides specialized messages which are more convenient to use. We will present one of them in this section and the remaining ones later.

Repeating a block a fixed number of times – `timesRepeat: aBlock`

If you want to flash a label in a window three times, you can use the following construct:

3 *timesRepeat*: ["statements to display the label"
"statements to create a short delay to keep the label displayed"
"statements to hide the label"
"statements to create a short delay to keep the label hidden"]

Method *timesRepeat*: is defined in the abstract class *Integer* because its receiver may be any kind of integer number, and its operation is described by the flowchart in Figure 5.3.

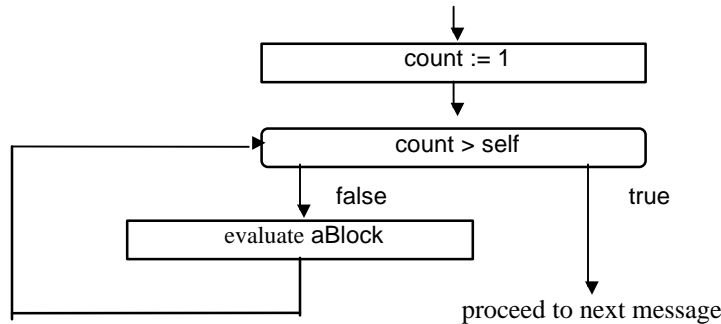


Figure 5.3. Execution of *n timesRepeat: aBlock*.

Example 1. Ringing the bell

All computers have a built-in beeper, conventionally referred to as the 'bell'. To beep the bell five times, execute

5 *timesRepeat*: [Screen default ringBell]

Class *Screen* is an interesting part of the library which knows various things about your computer, such as how many pixels its screen has, how to 'ring the bell', and how to perform interesting graphics operations. To access its single instance, send it the default message as above.

Example 2. Testing the quality of Smalltalk's random number generator

We have already mentioned that one of the motivations for object-oriented programming was simulation of problems such as queues in a bank. Problems of this kind are not deterministic because customers arrive at unpredictable times, and simulation thus requires tools for generation of random numbers. To satisfy this need, Smalltalk contains a class called *Random*.

Sending message *new* to class *Random* returns a new random number generator, a manufacturer of random numbers. Every time you send message *next* to the random number generator (Figure 5.4), it returns a floating-point number between 0 and 1. When you examine a sufficiently large collection of these random numbers, they should be distributed uniformly. In other words, if *I1* and *I2* are two arbitrary but equally long intervals between 0 and 1 and generate many random numbers, the number of samples falling into *I1* should be about the same as the number of numbers in *I2*. Although most applications require random numbers distributed in some non-uniform way, any distribution can be generated from uniform distribution and a uniform random number generator is thus a sufficient basis for all probabilistic simulations.

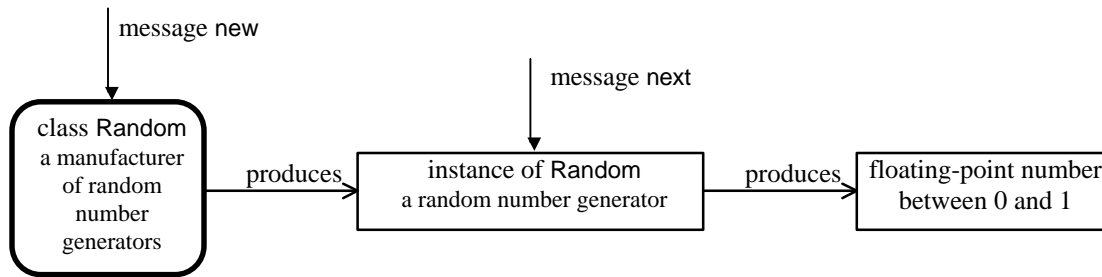


Figure 5.4. The use of class Random.

In this example, we will test how well Random accomplishes its purpose. We will select an interval I between 0 and 1, generate 10,000 random numbers, and count how many of them fall into I. We will then evaluate whether this number is appropriate for the width of the interval. As an example, if I is the interval from 0 to 0.1, its width is 1/10 of the 0 to 1 interval and the number of random numbers falling into I should be about 1/10 of the total number of generated samples. The test code is quite simple:

```

| count generator |
"Initialize count and create a random number generator."
count := 0.
generator := Random new.
"Run the test."
10000 timesRepeat: [ "Generate a random number and test whether it falls into our interval. If so,
                    increment the count."
                    ((generator next) between: 0 and 0.1)
                    ifTrue: [count := count + 1].
Transcript clear; show: 'The number of samples falling between 0 and 0.1 is ', count printString

```

When we executed this code, we got

The number of samples falling between 0 and 0.1 is 987

This is quite nice - the perfect answer would be 1,000 but we cannot expect that it will be achieved because the numbers are random. However, what if we were just lucky? Let's modify the program to repeat the whole test 10 times as follows:

```

| count generator |
"Initialize count and create a random number generator"
count := 0.
generator := Random new.
Transcript clear.
10 timesRepeat: "Do the whole test 10 times."
    [count := 0.
    10000 timesRepeat: 'How many of the 10,000 numbers fall into our interval?.'
    [((generator next) between: 0 and: 0.1) ifTrue: [count := count + 1]].
    Transcript show: 'The number of samples falling between 0 and 0.1 is ', count printString; cr]

```

The structure of this code fragment is a *nested loop* - one loop (10000 timesRepeat:) inside another (10 timesRepeat:). When Smalltalk executes the inner loop 1000 times, it checks whether the outer loop has been executed enough times. If not, it repeats the inner loop 1000 times again, checks again, and so on until the outer loop is executed 10 times (Figure 5.5).

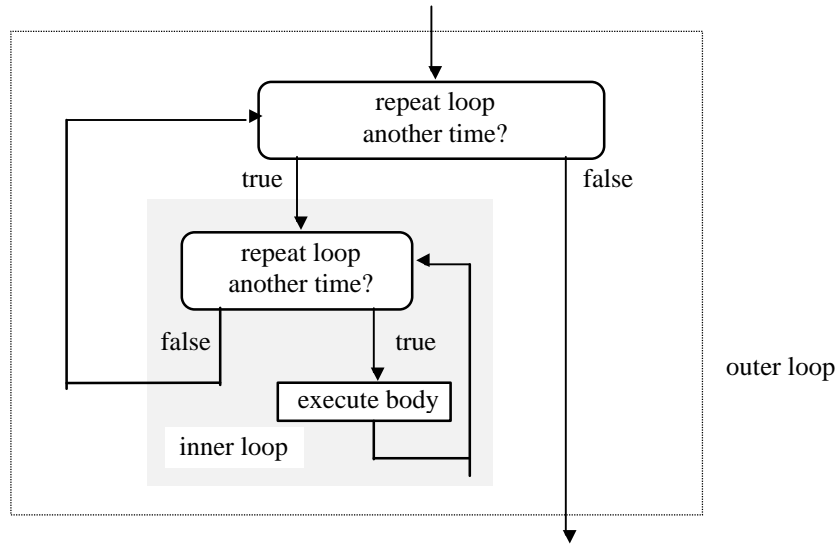


Figure 5.5. Nested loops.

When we executed this program, we got the following output:

The number of samples falling between 0 and 0.1 is 1001
The number of samples falling between 0 and 0.1 is 1006
The number of samples falling between 0 and 0.1 is 993
The number of samples falling between 0 and 0.1 is 1013
The number of samples falling between 0 and 0.1 is 976
The number of samples falling between 0 and 0.1 is 993
The number of samples falling between 0 and 0.1 is 1004
The number of samples falling between 0 and 0.1 is 1023
The number of samples falling between 0 and 0.1 is 1000
The number of samples falling between 0 and 0.1 is 1006

which is quite satisfactory. By the way, note that the numbers are different in each run. This is to be expected because the numbers are random.

Example 3. Definition of timesRepeat:

To conclude this section, let us examine the definition of timesRepeat:.

timesRepeat: aBlock

```
"Evaluate the argument, aBlock, the number of times represented by the receiver."  
| count |  
count := 1.  
[count <= self]  
  whileTrue: [aBlock value.  
              [count := count + 1]
```

The implementation is based on whileTrue: and so timesRepeat: is not strictly necessary because we can get the same effect by using whileTrue: directly. However, the solution with repeatTimes: is much neater. When you compare

```
5 timesRepeat: [Transcript show: 'Time to go home!'; cr]
```

with

```
count := 1.  
[count > 5] whileTrue: [Transcript show: 'Time to go home!'; cr. count := count + 1]
```


you will agree that the second form is longer (requiring more typing), more difficult to understand, and more error prone because we must remember to keep track of the count. (If we forget, we will get an infinite loop). Message `timesRepeat:` thus makes programming easier and a sufficient justification why this and many other apparently redundant messages are in the Smalltalk library.

Main lessons learned:

- To repeat a block of statements a known number of times, use message `timesRepeat:`.
- An iteration enclosed inside another iteration is called a nested iteration.
- If the Smalltalk library contains a message that does exactly what you need, use it. This will save you time re-inventing the wheel and making mistakes.

Exercises

1. Truly random numbers should never repeat the previously generated sequence but random number generators are based on arithmetic and produce repetitive sequences (they are thus properly called pseudo-random). The length of the sequence is called its period and a good pseudo-random number generator should thus have a large period. Is the period of `Random` large enough?
2. Class `Screen` defines various interesting methods. Explore and execute the following ones: `resolution`, `contentsFromUser`, `bounds`, `dragShape:...`, and `zoom:to:duration:`. Some of them contain example code.
3. Write a code fragment to print
 - a. one line containing ten * symbols (simple loop)
 - b. five lines, each containing ten * symbols (nested loops)
4. Modify the test of `Random` as follows:
 - a. Allow the user to specify how many random numbers should be generated.
 - b. Allow the user to specify the start and end points of the interval.
 - c. Allow the user to specify how many times to repeat the test.
 - d. Calculate the average number of samples falling into the interval over all tests.
5. Convert the random generator test into a test method in class protocol testing in class `Random`.
6. Some people think that the `timesRepeat: aBlock` message is not natural and that `aBlock repeatTimes: anInteger` as in `aBlock repeatTimes: 17` would be better. Define such a method and test it.
7. Write a summary of the main features of class `Random` and find all references to it in the library.

5.5 Repeating a block for all numbers between a start and a stop value

Method `repeatTimes:` is rarely used because it does not happen very often that you want to do exactly the same thing several times. Much more often, you need to do something with a value moving over a range - such as when you want to print a table of factorials from 1 to 100. The message that performs this task is `to:do:` and its basic use is as follows:

```
startNumber to: endNumber: do: oneArgumentBlock
```

where `oneArgumentBlock` is a block with one *internal argument*. The block is repeatedly executed with all successive values of the internal argument from `startNumber` to `endNumber` in increments of 1 (Figure 5.6).

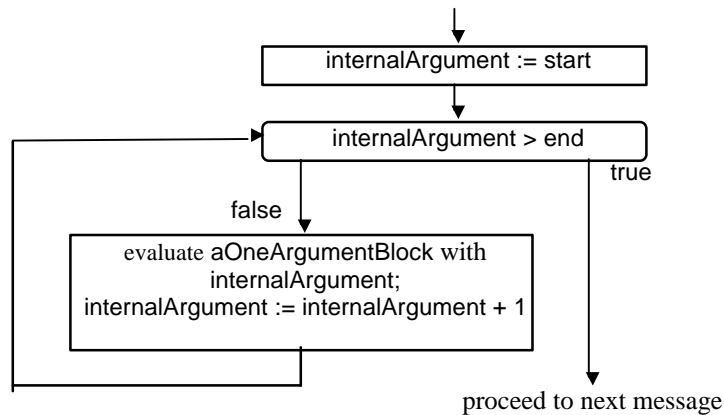


Figure 5.6. Execution of start to: end: do: aOneArgumentBlock.

Example 1: Print all numbers from 1 to 5 with their squares and cubes in the Transcript

Solution: Repeat the calculation and output for each value of the argument as it ranges through the interval:

Transcript clear.

1 to: 5 do:

```

[:arg | "arg is internal block argument. During the execution, it will assume values 1, 2, ..., 5."
  Transcript show: arg printString; tab;
  show: arg squared printString; tab;
  show: (arg squared * arg) printString; cr]
  
```

The output is

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

Blocks and arguments obey the following rules:

- A block may have any number of arguments. We have seen blocks without arguments such as the block used in ifTrue:, a method with one argument (to:do:) and we will see methods with more arguments later.
- All arguments defined for a block are listed immediately behind the opening bracket, each preceded by a colon. The whole list is followed by a vertical bar. A block with three arguments, for example, would be used as follows:

```
[:arg1 :arg2 :arg3 | "Body of block."]
```

- The name of block arguments must not clash with identifiers already declared in a surrounding scope. As an example, if a one-argument block is used in a method with a local variable or keyword argument called number, the block argument must not be called number. Similarly, if the block is nested inside another block, their argument names must not clash. The same applies to a conflict with the names of instance variables of the class. If any of these scoping rules are violated, Smalltalk displays a warning message as in Figure 5.7.

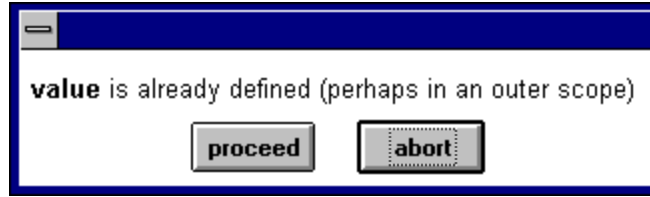


Figure 5.7. Block argument value is already defined in an outer scope.

- A block must not assign a new value to any of its arguments and any attempt to do so is caught by the compiler. (The same is true for method arguments. In fact, the concepts of a method and a block are very similar.)
- The role and nature of individual block arguments depends on the definition of the message that uses the block. In the case of `to:do:`, the block argument must be a number and its role is a counter or an index.

Before we close this section, we will now show how `to:do:` might be defined to illustrate how a method acquires a block argument. Our definition is different from the definition in the library because the library definition uses several concepts that will be covered later.

to: end do: aOneArgumentBlock

"For each number in the interval from the receiver to the argument, incrementing by 1, evaluate the block."

```
| index |  
    index := self.  
    [index <= end] whileTrue: [aOneArgumentBlock value: index. index := index + 1]
```

The critical part of the definition is message `value: index` which assigns `index` as the value of the block argument and causes evaluation of the block. In other words, the `value: message` is the one-argument-block equivalent of the zero-argument-block `value` message

The fact that the definition of `to:do:` evaluates `aOneArgumentBlock` with the `value: message` (rather than the `value` message) is the reason why you must use a one-argument block when you send `to:do:`. The number of arguments in a block is thus dictated by the block's use in the definition. The role of the argument, of course, depends on how the argument is used. In this case, the argument is evaluated with `index` and `index` is used to do arithmetic, so the argument must be a number.

As a more direct illustration of the operation of `value` messages, evaluating the zero-argument block in

```
[Transcript show: 'A string'] value
```

has the same effect as

```
Transcript show: 'A string'
```

whereas evaluation of the one-argument block in

```
[aString [Transcript show: aString] value: 'Block argument']
```

has the same effect as

```
Transcript show: 'Block argument'
```

and

```
[string1 : string2 [Transcript show: string1; show: string2] value: 'argument 1 ' value: 'argument2']
```

has the same effect as

Transcript show: 'argument 1 ' ; show: 'argument2'

Finally, a word of caution. If you search for all references to `value:` you will get a very long list but most of them refer to the `value:` method used with 'value holders' and have completely different meaning.

Main lessons learned:

- To execute a block for a range of values in increments of 1, use `to: anInteger do: aOneArgumentBlock`.
- The block in message `to:do:` has one internal argument.
- The receiver of `to:do:` may be any number.
- The number of arguments required in a block used as an argument of a message depends on the definition of the method in which the block argument is used.
- Use `value` to evaluate a zero-argument block, `value:` to evaluate a one-argument block, and `value:value:` to evaluate a two-argument block. Class `BlockClosure` defines additional value messages.

Exercises

1. List all numbers between 1 and 1,000 that are divisible by 11 and 13 in the Transcript. Use `to:do:`, `timesRepeat:`, and `whileTrue` and compare the complexity and readability of the code.

5.6 Repeating a block with a specified increment

Message `to:do:` lets you specify the start and end values but not the step (increment) between consecutive values of the block argument. Message `to:by:do:` allows you to specify the increment too (Figure 5.8). As an example,

```
1 to: 100 by: 2 do: [:number | Transcript show: number printString; cr]
```

prints all odd numbers from 1 and 99.

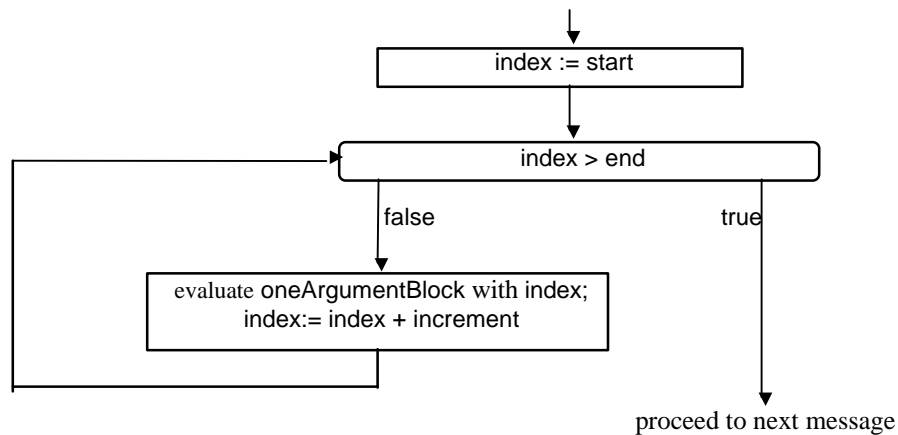


Figure 5.8. Execution of `start to: end by: increment do: oneArgumentBlock`.

All the numeric arguments of the message (start, end, and increment) may be any numbers - positive, negative, integer, floating-point, fractions, and so on. Don't forget, however, that floating-point arithmetic is inaccurate³ and using floating-point numbers with `to:by:do:` may produce unexpected results. The following example illustrates this point.

³ Smalltalk comments use the word of accuracy where others use the term precision.

Example 1: Print a table of base-10 logarithms for arguments from 1 to 10 in increments of 0.1

The problem is easily solve with the following expression:

Transcript clear.

```
1 to: 10 by: 0.1 do: [:number | Transcript show: number printString;
                        tab;
                        show: number log printString;
                        cr]
```

Unfortunately, the code stops short of outputting the logarithm of 10.0 because the accumulated inaccuracy of adding 0.1 causes the program to miss 10.0. To avoid this problem, we can use a Fraction increment as follows:

Transcript clear.

```
1 to: 10 by: 1/10 do: [:number | Transcript show: number asFloat printString;
                        tab;
                        show: number log printString;
```

This code produces the desired result but seems very slow - and so is, in fact, the original floating-point solution. This could have one of two reasons: either the calculation of logarithms is so slow, or the display is slow. In this case, the cause of the problem is the output and the solution is to delay output until the string is complete and then print it out all at once. To do this, use `nextPutAll: aString` to accumulate the individual strings in the Transcript object, and then flush to output:

Transcript clear.

"Calculate and gather the results."

```
1 to: 10 by: 1/10 do:[:number | Transcript nextPutAll: number asFloat printString;
                        tab;
                        nextPutAll: number log printString;
                        cr].
```

"Output the accumulated results."

Transcript flush

This version is much faster and the trick is worth remembering.

Example 2: How accurate is asRational?

In Section 1, we showed that `asRational` performs very accurate conversion from a floating-point number to a Fraction. We would now like to find how often the original number and the fraction converted back to a floating-point number are equal, how often they are different, and what is the maximum error when they are not different. As for the error, we are interested in the relative accuracy, the value of

$(\text{aNumber asRational asFloat}) - \text{aNumber} / \text{aNumber}$

To find the answer, we will repeat the conversion for all numbers between 1 and 10,000 in steps of 0.1, a total of 100,000 argument values.

Solution: With the stepwise iteration message that we just introduced, the solution is as follows:

| count maxError |

"Initialize."

count := 0. maxError := 0.

"Perform calculation."

1 to: 10000 by: 0.1 do:

```
    [:number | |error| (error := number asRational asFloat - number) isZero not
                ifTrue: [count := count + 1. maxError := (maxError max: error / number)]].
```

"Output the results."

Transcript show: 'Number of inaccurate conversions: ', count printString; cr;

show: 'Max error: ' maxError printString

When we executed this code, we got (after worrying for a little while whether we did not have an infinite loop)

Number of inaccurate conversions: 9357
Max error: 2.38419e-7

We conclude that asRational is very accurate because only 9,356 out of 100,000 calculations are inexact and this corresponds to about 0.9%. In other words, the conversion produces the same result as the original in 99% cases - at least in the chosen range. Moreover, when the results are not equal, the worst relative inaccuracy is only 0.00002%.

A few points are worth mentioning about the code fragment:

- We used a *temporary variable* error inside the block. You can declare any number of temporary variables inside a block whereas the number of block arguments is *given* by the value message used to evaluate the block. The temporary variable's scope is limited to the block.
- Blocks whose body does not depend on any external context are called *clean*, other blocks are either *copying* blocks or *full* blocks. As an example, a block using only internal temporary variables is clean whereas one that depends on externally declared local variables or arguments is not. Clean blocks are much more efficient, and temporary variables should thus always be declared inside blocks if possible. See the comment in BlockClosure for more details.

Main lessons learned:

- To iterate over a range of numbers, use to:do: if the step is 1, and to:by:do: if the step is not 1.
- For multiple output to Transcript, replace multiple show: messages with nextPutAll: and flush.
- Blocks may include any number of internal temporary variables.
- Blocks that don't depend on external context are called clean. They are more efficient than blocks that depend on external context, for example external temporary variables or method arguments.

Exercises

1. Print sin and cos values of arguments from 0 to $\pi/2$ in increments of 0.05 in the Transcript.
2. Repeat Exercise 1 but print results in reverse order of arguments.
3. Print a table of all multiples of 13 between 1 and 1,000 using to:by:do:
4. Run the test of asRational for a different range of values to check whether our result is representative. Repeat for Double numbers.
5. Which of the following combinations is correct?
 - a. method1: [:arg1 arg2 |] "Definition of method1 uses value:value:"
 - b. method1: [:arg1 :arg2 |] "Definition of method1 uses value:"
6. Method to:do: is a specialization of to:by:do: that takes 1 as increment. This kind of specialization is very common and class Dialog contains several other examples. List how many such specializations occur in its file name dialogs protocol and comment on their implementation.

5.7 Measuring the speed of arithmetic and other operations

After implementing the code of an application, we may want to optimize its critical parts to achieve satisfactory response time. To do this, we need some means of measuring how much time a piece of code requires to run, and where it spends most time. This activity is called *profiling* and the Advanced Tools extension of VisualWorks provides tools to perform such analysis. For simple measurements, a method in class Time gives useful answers too and this is the approach that we will use in this section.

Example 1. Comparing addition of different kinds of numbers

When we talked about different kinds of numbers, we said that some of them perform arithmetic faster than others. We will now do a little experiment to compare the speed of addition of `SmallInteger`, `LargePositiveInteger`, `Float`, `Double`, and `Fraction`. We will do this by packing addition into a block and measuring how long the block takes to execute. This is easy because class `Time` knows what time it is and to measure how long a block of code requires to execute, we just need to record the time before we start (time `t1`), the time just when the block is finished (time `t2`), and then subtract `t2-t1` to get the result. Although this idea is correct, it would not work because measuring time in seconds is too long for calculations. For this purpose, we will thus replace `Time` now with `Time millisecondClockValue` which returns a millisecond count of the computer (not derived from `Time now`). The solution is thus

```
| startTime endTime runTime |
startTime := Time millisecondClockValue.
"Execute the code here."
endTime := Time millisecondClockValue.
runTime := startTime - endTime
```

In fact, measuring the length of execution is so useful that class `Time` includes a method called `millisecondsToRun:` that performs this task. It takes a block as its argument and returns the number of milliseconds that it takes to execute. As an example,

```
Time millisecondsToRun: [x := 2 + 3]
```

returns the number of milliseconds required to add 2 and 3 and assign the result to variable `x`. The definition of `millisecondsToRun` is

millisecondsToRun: timedBlock

"Answer the number of milliseconds `timedBlock` takes to return its value."

```
| initialMilliseconds |
initialMilliseconds := self millisecondClockValue.
timedBlock value. "Evaluate the block argument here."
^self millisecondClockValue - initialMilliseconds
```

Now that we have the tool for measuring execution time, we must devise a strategy for getting a good answer to our question. The obvious approach is to execute addition for a pair of small integers and measure how long it takes, do this again for large integers, and so on, something like

```
| t |
Transcript clear.
"Do SmallInteger first"
t := Time millisecondsToRun: [2+3].
Transcript show: 'The time needed to add two small integers is ', t printString, ' milliseconds'; cr.
"Do LargePositiveInteger next."
t := Time millisecondsToRun: [20000000000+30000000000].
Transcript show: 'The time needed to add two large integers is ', t printString, ' milliseconds'; cr.
"And so on for other kinds of numbers."
```

Unfortunately, this will not work because addition of two integers takes only a few CPU cycles and modern CPUs running at 100 million cycles per second or more take only microseconds rather than milliseconds to execute them. To get a realistic value in milliseconds, we must do the addition many times. We should thus do something like

```
Time millisecondsToRun: [ 10000 timesRepeat: [2+3]]
and so on
```

This is not perfect either because it measures the time to execute `2+3` plus the time needed to execute the `repeatTimes:` message. To eliminate this effect, we will correct the obtained time by subtracting the time needed to run the `repeatTimes` message by itself as in

```
t1 := Time millisecondsToRun: [ 10000 timesRepeat: [2+3]].  
t2 := Time millisecondsToRun: [ 10000 timesRepeat: ["nothing"]].  
timeToAddIntegers := t1 - t2
```

Our complete solution is as follows:

```
| t1 t2 |  
Transcript clear.  
"SmallInteger: Two small integers."  
t1 := Time millisecondsToRun: [ 10000 timesRepeat: [2+3]].  
t2 := Time millisecondsToRun: [ 10000 timesRepeat: []].  
Transcript show: 'The time needed to add two small integers is ',  
                ((t1-t2) / 10000) asFloat printString, ' milliseconds'; cr.  
"LargePositiveInteger: Two large integers."  
t1 := Time millisecondsToRun: [ 10000 timesRepeat:  
    [20000000000+30000000000]].  
Transcript show: 'The time needed to add two large integers is ',  
                ((t1-t2) / 10000) asFloat printString, ' milliseconds'; cr.  
"and so on."
```

When I executed this program on my laptop computer, I got the following result:

```
The time needed to add two small integers is 1.0e-4 milliseconds  
The time needed to add two large integers is 0.0041 milliseconds  
The time needed to add two very large integers is 0.0075 milliseconds  
The time needed to add floating-point numbers is 9.0e-4 milliseconds  
The time needed to add two fractions is 0.0325 milliseconds
```

which confirms our expectations: Floats take longer than integers, large integers take longer than floats and the time depends on the size of the number because it requires repeated addition over all consecutive parts. Fractional arithmetic takes very long, even longer than arithmetic on very large integers.

If you repeat this experiment, you should consider that I have a standalone computer that I don't share with anybody. If you are using a CPU shared over the network, the result may be completely misleading because your CPU may be switching your work and somebody else's work during the execution of your program and the test will include the total elapsed time. But even if you have a standalone computer, the result may not be correct because the computer may be doing some other work during the calculation such as collecting unused objects (garbage collection).

More sophisticated profiling tools do not measure the time but instead sample program execution, taking a peek once every little while and keeping count of how many times each message is encountered. In the end, they produce the percentage of time spent in individual messages. You can then focus on the messages that are sent most often. Profiling tools can also tell you about the size of your objects.

Example 2 - Evaluating the speed of recursion

Recursion sometimes very naturally solves a problem but it is generally considered inefficient and undesirable. Besides, it tends to use more memory space than other solutions. A classic example of a problem that lends itself very well to recursive solution is the calculation of the *factorial* whose recursive definition is as follows:

The factorial $n!$ of an integer n is

- *undefined for $n < 0$*
- *1 if $n = 0$*
- *$n * (n-1)!$ if $n > 0$*

According to this definition, the factorial of 3 is 3 times the factorial of 2, which is 2 times the factorial of 1, which is 1. Altogether, the factorial of 3 is thus $3*2*1$.

VisualWorks library already contains a definition of factorial and this definition is based on iteration since $n! = n*(n-1)*(n-2)* \dots * 2 * 1$. In essence, the library definition is as follows:

factorial

```
| tmp |
self < 0 ifTrue: [^self warn: 'Factorials are defined only for non-negative integers'].
tmp := 1.
2 to: self do: [:i | tmp := tmp * i].
^tmp
```

However, the recursive definition can be more easily implemented as follows:

factorialRecursive

```
self < 0 ifTrue: [^self error: 'Negative argument for factorial']. "Open exeception window."
self = 0 ifTrue: [^1].
^self * (self - 1) factorialRecursive
```

We will now examine whether recursion is slower and if so, how much slower. We will use the technique from Example 1 as in the following code fragment:

```
| t |
Transcript clear.
t := Time millisecondsToRun: [1000 factorial].
Transcript show: 'Nonrecursive factorial. Time: ', t printString; cr.
t := Time millisecondsToRun: [1000 factorialRecursive].
Transcript show: 'Recursive factorial. Time: ', t printString; cr
```

When I executed this code, I obtained the following results

```
Nonrecursive factorial. Time: 131
Recursive factorial. Time: 190
```

and concluded that although there is a difference, it is smaller than I expected. We will thus analyze the code and try to find the explanation.

Both methods begin with the same test and the essence of the remaining calculation is multiplication of intermediate results. In the case of factorial, this is followed by incrementing the number which should be rather negligible with respect to multiplication, and another iteration. Could it be that multiplication is the culprit? Could it be that multiplication takes so long that it masks everything else and that our tests actually mostly measure the duration of multiplication rather than the effect of recursion? This is quite possible, especially when the numbers become `LargePositiveInteger` (which they do very quickly) - we have already seen how slow their arithmetic is.

To test this hypothesis, we will do recursion with a block that takes much less time - calculation of the sum of numbers from 1 to n using a recursive approach based on

$\text{sum}(n) = \text{sum}(n-1) + n$

and an iterative approach based on

```
1 to: n do: [:number | sum := sum + number]
```

We will not use the obvious solution

$\text{sum}(n) = n*(n+1)/2$

because our goal is to compare recursion with iteration doing over the same operation.

When we defined a sum method corresponding to these two styles (left as an exercise) and ran the following test

```
| t |  
Transcript clear.  
t := Time millisecondsToRun: [10000 sum].  
Transcript show: 'Nonrecursive sum. Time: ', t printString; cr.  
t := Time millisecondsToRun: [10000 sumRecursive].  
Transcript show: 'Recursive sum. Time: ', t printString; cr
```

the result was much more like what we expected and confirmed our hypothesis:

```
Nonrecursive sum. Time: 2  
Recursive sum. Time: 22678
```

This leads to the following conclusion: Recursive programs generally execute slower than their iterative counterparts. However, when the calculation performed during each recursive step is time-consuming, the difference between recursion and iteration may be negligible. A corollary is that if the calculation is substantial, converting a natural recursive solution to a less obvious iterative solution may not be worth the trouble. The elegance of recursive solutions makes them easier to understand which contributes to the maintainability of code. This sometimes makes recursive solutions preferable in complex problems, particularly in earlier stages of development before the code is optimized for speed.

Example 3: Observing recursion in action

To complete this section, execute the following code fragment

```
self halt.  
5 factorialRecursive
```

and follow the execution of the recursive definition of the factorial all the way from 5 to 0 by using *send* in the Debugger. When *self* decrements from the initial value of 5 to the end value of 0, the debugger window is as in Figure 5.9, showing all the stacked recursive calls. Recursion then begins to unwind, completing the messages piled up on the stack, and eventually returning to the original program fragment with the final result. This little experiment shows very nicely the nature of recursion.

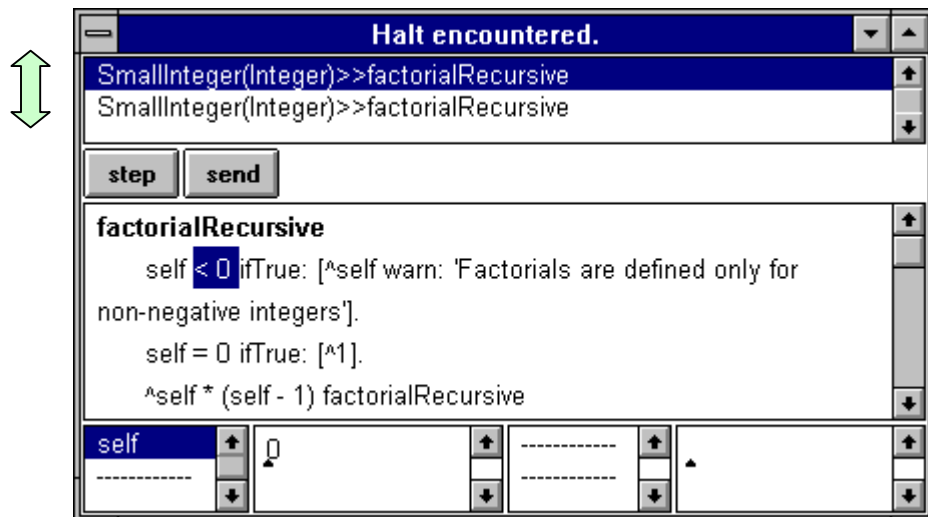


Figure 5.9. The top two levels of the message stack during the execution of 5 factorialRecursive.

Main lessons learned:

- The first implementation of an application should not focus on efficiency. Speed should only become a concern when the running application is too slow.
- In trying to improve speed, focus on those parts of the code where the application spends most of its time.
- Determining where the application spends most of its time is called profiling.
- Profiling can also be used to check how much memory an application uses and where.
- Analyzing code for efficiency requires careful analysis because the most natural conclusions may be incorrect.
- Recursion requires overhead but this overhead may be outweighed by calculation required in each recursion step.
- Solving problems with recursion is sometimes much easier than using other approaches and recursive solutions are often preferred until final speed optimization.

Exercises

1. Method `timesRepeat:` is implemented by the `whileTrue:` message and so `timesRepeat:` must be slower. How much slower is it?
2. Test whether our precautions in Example 1 (eliminating the effect of repetition) were justified.
3. Add Double arithmetic to Example 1 and compare with Float.
4. We noted that the duration of addition of large integers depends on their size. Obtain an experimental dependency between the duration of addition and the length of the operands, and explain it qualitatively. To do this, create some large integers such as 100 factorial, 500 factorial, and 1000 factorial, store them in variables `x`, `y`, and `z`, and perform `x+1` and `y + 1` and `z + 1` (without assignment) a sufficient number of times, measuring the speed. Print the results along with some indication of the size of the numbers, draw a graph of the dependency, and explain it.
5. Compare the speed of `+`, `-`, `*` and `/` in the most important number classes.
6. Write recursive and non-recursive definitions of the methods used in Example 3 and test their relative speed.
7. About 2,000 years ago, the Greek mathematician Euclid invented the following algorithm for calculating the greatest common divisor `gcd(m,n)` of two positive integers `m` and `n`:
If `m = n`, `gcd = m`
If `m < n`, `gcd(m,n) = gcd(n,m)`
If `m > n`, `gcd(m,n) = gcd(m-n,n)`
Use this recursive definition to define method `gcdRecursive:` to calculate `gcd` recursively.
8. VisualWorks library contains a built-in `gcd:` method. Compare its speed with the speed of the recursive implementation. Analyze the result.
9. The Fibonacci series is defined as follows: `Fib(n) = 1` for `n = 1, 2`, `Fib(n-1) + Fib(n-2)` for `n > 2`, and `Fundefined` for all other values of `n`. Write and test method `fibonacci` to calculate Fibonacci numbers.

5.8 A new class: Currency

In this section, we will define a new class as a simple application of numbers. In doing this, we will introduce new techniques useful for testing.

When we discussed number classes, we noted that `FixedPoint` was designed as a possible solution of the problem of representing currency. The problem is not as simple as it might seem, for example because operations such as conversion from one currency to another are very sensitive to tiny inaccuracies whose effect becomes significant when we deal with billions of dollars or convert back and forth between currencies.

We will now define an alternative simple `Currency` class for currencies whose units consist of dollars and cents. Our solution is far from perfect and we leave the fine points of currencies to experts.

Design

The behaviors that we expect of Currency include creation (create a new Currency object), arithmetic (add or subtract two Currency objects), comparison (equal, not equal, less than, and so on), and printing (for testing and inspecting). Each of these behaviors will be implemented as a protocol, and the notion of dollars and cents suggests the use of two instance variables called dollars and cents.

We have now decided all the major features of Currency except where to put it in the class hierarchy. Currency objects are somewhat like numbers, so they will be somewhere on the Magnitude branch of the class tree. Since currencies can do simple arithmetic, we must consider whether the superclass should be ArithmeticValue or even Number. However, although we want to be able to add and subtract Currency objects, we don't need any other arithmetic and in this sense, Currency objects are thus distinct from numbers. Also, the comment of ArithmeticValue says that its subclasses must implement certain arithmetic messages needed by some other messages, and since we don't really need most of them, this reinforces our opinion that Currency should not be a subclass of ArithmeticValue. Most importantly, however, currency objects are conceptually different from numbers and it does not make sense to think of them as specialized numbers.

We conclude that making Currency a subclass of ArithmeticValue or Number has too many disadvantages and few advantages and we will thus define it as a subclass of Magnitude, inheriting certain comparison messages. In this, we will follow in the footsteps of those who defined classes Time and Date. We will put Currency into the existing category Tests as we did with class Name.

We are now ready to define the new class. Start by opening the browser on category 'Tests', edit the class template as in

```
Magnitude subclass: #Currency
  instanceVariableNames: 'dollars cents'
  classVariableNames: ''
  poolDictionaries: ''
  category: Tests'
```

and execute *accept* from the <operate> menu of the text view.

As the next step, add a comment explaining the purpose of the class, its instance variables, and any special notable features. Our comment for Currency will be as follows:

I implement a decimal currency such as dollars and cents.

Instance variables:

dollars <Integer> - the dollar part of the currency
cents <Integer> - the cent part of the currency

We are now ready to implement the behaviors. Since we cannot do anything unless we can create an instance, we will start with the creation message.

Creation

The style that we want to use for the creation message is to specify initial values for dollars and cents as in

Currency dollars: 100 cents: 45

Message dollars:cents: is obviously a class message because its receiver is Currency, and its implementation will follow Smalltalk's established pattern for creation messages:

1. Create a new Currency object by sending new to the Currency class. This inherited message creates an uninitialized instance of Currency with its dollars and cents instance variables set to nil.
2. Send dollars: anInteger to this instance, setting its instance variable dollars to the value of the argument. (Accessing method dollars: does not yet exist and we will have to define it.)
3. Send cents: anInteger to the Currency object, setting its instance variable cents to the value of the argument. (We will again have to define method cents: first.)
4. Return the new Currency object.

The definition written along these lines is as follows:

dollars: dollarsInteger cents: centsInteger

"Creates a new initialized Currency object."

^ self new

dollars: dollarsInteger;
cents: centsInteger

Note the following points:

- The effect of self new is equivalent to Currency new but the self new version is preferable: If we created a subclass of Currency such as NewCurrency, the version using self new would create an instance of NewCurrency (which is what we would probably expect) whereas the version using Currency new would create an instance of Currency (probably not what we want). Conclusion: Don't refer to a class explicitly unless you must.
- Although the inherited version of new is defined in class Behavior, it creates an instance of Currency or whatever class sends the message.
- It is a very common mistake to forget the return operator ^ in a creation method. The method then returns the class rather than the new instance.
- Use cascading to send initialization messages to the new instance of Currency.

Although we could now enter the definition of dollars:cents: into the browser and compile it before creating dollars: and cents:, it will be better to define accessing methods dollars: and cents: first. Both are instance messages because their receiver is self new - an instance of Currency - and they simply assign new values to instance variables:

dollars: anInteger

dollars := anInteger

and similarly for cents:.

The methods are so trivial that they don't require a comment. Create these instance methods and the dollars:cents: creation method, and test whether everything works by executing

```
Currency dollars: 100 cents: 47
```

with *inspect*. When you examine the values of instance variables, you will find that everything is OK.

Printing

At present, the only way to see the values of instance variables of Currency objects is to open an inspector or to access them. Our life would be much easier if we could display the components of Currency by *printString* but this would, of course, only produce the string 'a Currency'. If we want *printString* to produce useful information about instances of a new class, we must change the *printString* mechanism and to do this, we must understand how it works.

When we browse the implementers of *printString* we find that there is essentially only one - class Object - and that the definition reads as follows:

printString

```
"Answer a String whose characters are a description of the receiver."  
| aStream |  
aStream := WriteStream on: (String new: 16).  
self printOn: aStream.  
^aStream contents
```

The critical message is obviously *self printOn: aStream* and to change the behavior of *printString*, we must therefore redefine *printOn:*. There are many implementors of *printOn:* because this method customizes *printString* to produce useful information. As an example, the definition in class Contract is as follows:

printOn: aStream

```
aStream nextPutAll: self class name, ' as ', self server name, ' :: ', name printString; cr
```

and we will use it as a template even though we don't understand very well yet what it does. First, however, we must decide how we want a Currency to display itself. We selected the following style:

```
Currency dollars: 100 cents: 47
```

executed with *print* it (thus using *printString*) should produce

```
a Currency 100 dollars, 47 cents
```

To produce this format, our *printOn:* message must execute the following steps:

1. Output the name of the class into *aStream* (we don't care what a Stream is, at this point).
2. Output the dollars part followed by ' dollars '.
3. Output the cents followed by ' cents'.

According to this algorithm and the *printOn:* template, we define *printOn:* in class Currency as follows:

printOn: aStream

```
"Append to the argument, aStream, the description of a Currency."  
aStream nextPutAll: self class name,  
self dollars printString,  
' dollars, '  
self cents printString,  
' cents, '
```

To test how this method work, we execute

Transcript show: (Currency dollars: 13 cents: 27) printString

and we indeed obtain the desired output.

Note again that we did not need to understand the details, such as what a *Stream* is, in order to write a working definition. This is quite all right - don't try to understand every object that you must use; all you need is to know how to use it. A carpenter also does not need to know what is inside a drill if he wants to use one, and you don't care about the internal operation of the mail system if you want to send a letter. (On the other hand, it never hurts to understand your tools and if you don't have any pressing matters on your agenda, go ahead and learn about a new class.)

It is interesting to note that *printString* is also used by the inspector to display *self* and when you execute

Currency dollars: 13 cents: 27

with *inspect*, you will again obtain the desired information under the *self* entry in the inspector window.

Arithmetic and comparison

We will restrict ourselves to addition and leave subtraction to you. To add two currencies, we will create a new *Currency* object and set its *dollars* part to the sum of the *dollars* parts of the receiver and the argument, and the *cents* part to the sum of the *cents* parts as follows:

+ aCurrency

^Currency dollars: (self dollars + aCurrency dollars) cents: (self cents + aCurrency cents)

We assume that you defined the accessing messages *dollars* and *cents*. Add the *+* method to the arithmetic protocol of *Currency* and test it by executing

(Currency dollars: 100 cents: 47) + (Currency dollars: 25 cents: 32)

with *inspect* or *print it*. The result shows that everything is OK⁴.

Finally the *comparison* protocol. We already know that some comparison methods are defined in class *Magnitude* and we will now check its comment to find which comparison methods must be defined so that we can inherit the rest. The relevant part of the comment of *Magnitude* is

Subclasses must implement the following messages:

<
=
hash

We will start with equality. Two *Currency* objects are equal if they have the same number of dollars and the same number of cents, hence

= aCurrency

^(self dollars = aCurrency dollars) and: [self cents = aCurrency cents]

Our next target is the *hash* method. Its purpose is to assign an integer value to an object, such that can be used as its shorthand representation, something like a student number. Just like a student number or any other id, each object should have its unique hash value to distinguish it from other similar objects. This may be difficult and so hashing follows a weaker rule, namely that equal objects should have the same hash value. Because of this, and because some very important operations in Smalltalk use hashing instead of equality, redefining equality should always be accompanied by redefining hash.

⁴ If you have reservations about our approach, wait for the improved implementation later.

A hash method is already defined in class `Object` but since we redefined equality, we will also redefine `hash`. Our class has two internal variables and we will thus borrow a hash definition from another class with two variables - class `Fraction` - and convert it for our needs. Our definition will be

hash

`^dollars hash bitXor: cents hash`

We leave the definition of `<` to you as an exercise.

Main lessons learned:

- When constructing new classes and methods, start with creation, accessing, and printing protocols and test your definitions immediately.
- The recommended style for creating a new instance is to initialize its instance variables.
- Don't refer to a class explicitly unless you must.
- A very frequent mistake in writing a creation method is to forget the return operator. The method then returns the receiver class rather than the new object.
- When you need to define a specialized version of an existing method, check how other classes do it and follow the template. You don't have to understand the details.

Exercises

1. Add class protocol testing and write methods to perform complete testing of `Currency`.
2. As an alternative to the previous exercise, remove testing from `Currency` and create a new class called `CurrencyTesting` containing test methods in a class protocol. The `Currency` objects used in the tests can be stored in class variables and the class variables initialized in the *class* initialization method `initialize`.
3. Write and test the `-` method for subtraction of currencies.
4. Write and test the implementation of `<`. What about `<=`, `>`, and `>=`?
5. In the definition of `+ aCurrency`, we had to access the components of `aCurrency` with accessing methods `dollars` and `cents`. The `dollars` and `cents` instance variables of the *receiver*, however, could have been accessed directly. Many Smalltalk experts argue that instance variables should always be accessed by accessing methods even though it is not strictly necessary. Give one argument supporting this position and one argument against it.
6. Define a new class called `UniformRandom` that allows the user to create a random number generator for numbers uniformly distributed in any interval, not just between 0 and 1 as `Random` does. The creation message should be `UniformRandom from: startNumber to: endNumber` and the instance message for creating a new random number should be `next`. (Hint: It is tempting to make `UniformRandom` a subclass of `Random` because they are so closely related. However, `UniformRandom` would not inherit any useful functionality or components from `Random`, it would have to redefine `next`, and its creation message is different - so there is no point in subclassing. Since there is no related class in the Smalltalk hierarchy, make `UniformRandom` a subclass of `Object` and define it with instance variables `generator` - an instance of `Random` to be used to calculate random numbers in `<0,1>` - and `start` and `end` to hold the endpoints of the interval. Calculations of new random numbers will depend on getting a new random number from `generator`, and scaling it using the `start` and `end` values of the interval. As we mentioned before, this approach is called *delegation* because `UniformRandom` passes responsibility for a part of its functionality to one of its components. *Delegation is generally preferred to inheritance which should be used only in cases of pure specialization.*)

5.9 Another implementation of Currency

Our existing implementation of `Currency` is fine in terms of its protocols but unsatisfactory in implementation. As an example, expression

Currency dollars: 100 cents: 3412

creates a Currency object with 100 dollars and 3412 cents but we would prefer a Currency object with 134 dollars and 12 cents. Similarly,

Currency dollars: 100 cents: -34

creates a strange object that does not make sense, and our comparison message = returns false for

(Currency dollars: 100 cents: 3412) = (Currency dollars: 134 cents: 12)

whereas we would probably expected true.

If we had thought about the implications of our design more carefully (in other words, if we did the design properly), we would have discovered the problem and changed the design before implementing it. At this point, we will have to make some changes. One possible approach is to leave the protocols, and redefine Currency in terms of cents only and eliminate dollars. Let's do this and see what implications it has for the methods that we had defined.

Creation

We still want to be able to create Currency objects from dollars and cents and so our creation message must convert dollars to cents. The new version of the creation method is simply

dollars: dollarsInteger cents: centsInteger

"Creates a new initialized Currency object."

^ self new cents: centsInteger + integer1 * dollarsInteger

Accessing

We don't really need method dollars: (it was only needed for creation) but we will leave it in because we don't want to change the message interface, in case somebody wrote another class that depends on it. We will leave it to you to redefine it to change dollars to cents. Methods cents and dollars must also be changed, as in

dollars

"Return dollar part of amount."

^ cents quo: 100

and

cents

"Return cents part of amount."

^ cents \\ 100

Arithmetic

The old implementation is

+ aCurrency

^Currency dollars: self dollars + aCurrency dollars cents: self cents + aCurrency cents

but with the new creation method cents:, we can define addition much more easily as follows:

+ aCurrency

^Currency cents: self cents + aCurrency cents

The only problem is that we don't have a creation message called cents: but this is easy to fix:

cents: centsInteger

"Creates a new initialized Currency object."
^self new cents: centsInteger

Note that we now have two different cents: methods but that's OK because one is a class method and the other is an instance method. This means that Currency cannot be confused because only one cents: method exists for it, and instances cannot be confused either.

Comparison

Comparison also becomes much simpler - and we don't have to worry about cents above 100 or below -100

= aCurrency

^self cents = aCurrency cents

The hash method can use the hash value of integer cents as follows:

hash

^cents hash

Printing

We will leave this protocol to you as an exercise.

This completes our redesign. The changes that we made are very small and simplify the code substantially. Note that they don't affect any other code that may use Currency because the old message interface remains, both in form and in effect. In particular, the test methods suggested in exercises at the end of the previous section will also work and can be used to test the new implementation. The principle of information hiding thus allowed us to reimplement Currency without any effect on any applications that may already be using it. (Note, however, that if we created some Currency objects using the first implementation and stored them in a file, we could not reuse them with the new implementation without some additional conversion.)

Main lessons learned:

- If we modify a class without changing its message interface, objects already using the class will not be affected. This is a consequence of information hiding.

Exercises

1. Complete the new implementation of Currency. Don't forget to change the comment and to retest the new implementation.
2. Compare our implementation of Currency with FixedPoint with scaling factor 2.
3. In an article in Smalltalk Report, Kent Beck described an implementation of currency that goes beyond dollars and cents and avoids conversion inaccuracy as much as possible. His approach is based on the fact that a wallet might contain any combination of currencies such as US \$3.50, Canadian \$6.75, and Japanese ¥300. This combination does not automatically convert to one currency unless explicitly required. Implement a Currency class based on this idea but restricted to four currencies: Canadian dollars, US dollars, British pounds, and French francs. When an amount is added or subtracted, arithmetic is done only on equal denominations and conversion is performed only when explicitly required.

4. Currency is a class that operates on objects measured in units. Develop a class called `Length` to handle the metric system with millimeteres, centimeters, meters and kilometers, and the British system with inches, feet, yards, and miles. Use the same approach as in the previous exercise.
5. Repeat the previous exercise for weight objects in units of grams, dekagrams, and kilograms on one side, and ounces and pounds on the other.

5.10 Generalized rectangles

As another example of a new class, we will now implement a new class for rectangles. One rectangle class, called `Rectangle`, is already in the library but it is restricted to rectangles with horizontal and vertical sides because its intended use is for windows on the screen. Our class will handle any rectangles and we will thus call it `AnyRectangle`. Before we start developing it, let's decide what we want to do with it and what functionality it should have.

Our reason for adding `AnyRectangle` is that rectangles are fairly common geometric objects and that we might want to use them in a drawing program and similar applications. The main purpose of the new class is thus to support display and graphics operations such as *move* and *rotate*. Purpose determines functionality and we thus decide on the following protocols:

- creation
- accessing - get and set selected rectangle parameters
- transformation - move and rotate
- display - drawing on the screen
- printing - for testing purposes

The next question is how to represent a rectangle. The horizontal/vertical `Rectangle` class uses the coordinates of the upper left and lower right corner because the orientation of the sides is known. In our case, two corners are not enough because there is an infinite number of rectangles that have the same two corners. As an example, if the two given corners are C1 and C2 in Figure 5.10, any rectangle whose third corner lies on a circle with diameter C1 and C2 is acceptable.

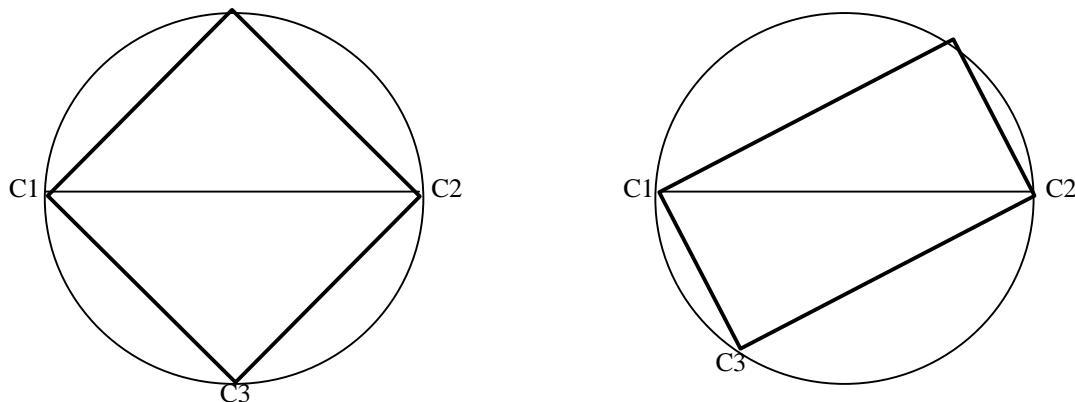
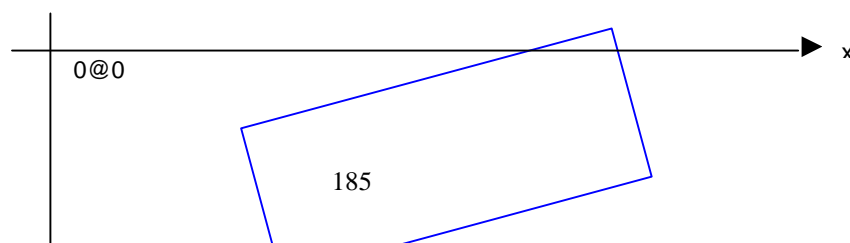


Figure 5.10. Two corners, such as C1 and C2, don't define a unique rectangle.

Although two corners don't identify a rectangle, three corners do. However, three points don't necessarily define a right angle and so three points are not automatically corners of a rectangle. We will thus represent a rectangle by its two opposite corners and an angle, thinking of the rectangle as a rotated horizontal-vertical rectangle (Figure 5.11).



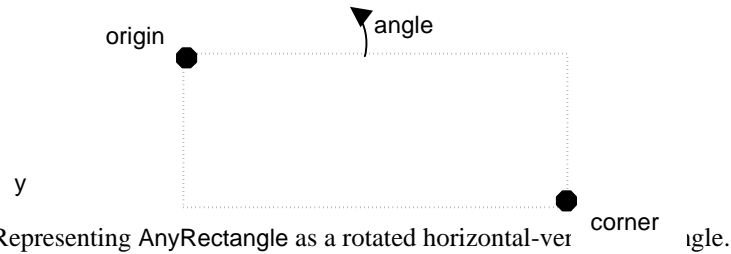


Figure 5.11. Representing AnyRectangle as a rotated horizontal-vertical rectangle.

Having made this decision, we can now start implementing the methods. We will use `Rectangle` as the superclass of `AnyRectangle` because it already defines all of the protocols we need, but we will leave it to you to check whether all inherited protocols remain valid as an exercise.

Creation. Since a rectangle is defined by three instance variables, the creation method will have three keywords. It will create a new instance and then send instance messages to assign instance variable values:

```
origin: point1 corner: point2 angle: radians  
"Create a rectangle with three corners."  
^self new origin: point1; corner: point2; angle: radians
```

where the instance accessing methods are standard setter methods. A typical creation message would look like

```
AnyRectangle origin: 1@100 corner: 11@90 angle: 3.14/4
```

Transformation.

Moving a rectangle means shifting the x coordinate and the y coordinate of each corner without changing the angle. This means that `AnyRectangle` is moved the same way as an ordinary `Rectangle` and the inherited methods `moveTo:` and `moveBy:` should thus work without change.

Rotation. Before we start thinking about implementation, we must decide what kind of rotation we want. First, we will assume that we perform rotation by keeping the upper left corner (inherited instance variable `origin`) fixed and rotating the rectangle around it. In principle, we can see two ways of doing this: by specifying the angle of rotation, and by dragging the opposite corner interactively.

The first kind is easy – we just increment the angle as follows:

```
rotateByAngle: aNumber  
angle := angle + aNumber
```

If we want to lay a foundation for letting the user rotate the rectangle interactively, the task gets a bit harder and we must do some trigonometry. The logic of the operation is as follows:

1. Calculate the angle of rotation from the new position of the dragged corner.
2. Change the angle instance variable by the calculated rotation angle.

We leave it to you to figure out the details but remember that the screen coordinate system measures x from left to right and y *from top to bottom*. Thanks to predefined methods for calculating distance and similar parameters, this problem is easier than you might think.

Printing. From the section on `Currency`, we know that if we want a new `printString` behavior, we must redefine the `printOn:` method. There is no reason to make the string for `AnyRectangle` much different from the string for `Rectangle` (the only difference is that we need add the angle variable) and so we will reuse `printOn:` from superclass `Rectangle`

printOn: aStream

"Append to the argument `aStream` a sequence of characters that identifies the receiver. The general format is `originPoint corner: cornerPoint.`"

```
origin printOn: aStream.  
aStream nextPutAll: ' corner: '.  
corner printOn: aStream
```

and extend it as follows:

printOn: aStream

"Append to the argument aStream a sequence of characters that identifies the receiver. The general format is originPoint corner: cornerPoint angle: angle."

```
super printOn: aStream.  
aStream nextPutAll: ' angle: '.  
angle printOn: aStream
```

In this definition, *super* is a special identifier that allows us to access an identically named method defined higher up in the class hierarchy. We will have more to say about it in Chapter 6.

Displaying. We will draw the rectangle by drawing the individual sides one after another. The subject of drawing will not be covered until Chapter 12 and we will thus implement this method by imitating the display method in class *LineSegment* which is defined as follows:

displayStrokedOn: aGraphicsContext

"Stroke the receiver on the supplied GraphicsContext."
aGraphicsContext displayLineFrom: start to: end

In this context, 'stroking' means simply drawing, and we don't need to know what a *GraphicsContext* is at this point. The method for displaying *AnyRectangle* must calculate the corners and draw the four lines. The whole definition is

displayStrokedOn: aGraphicsContext

```
"Stroke the receiver on the supplied GraphicsContext."  
| topRightCorner bottomLeftCorner bottomRightCorner |  
"Calculate corners."  
topRightCorner := self topRightCorner.  
bottomLeftCorner := self bottomLeftCorner.  
bottomRightCorner := topRightCorner + bottomLeftCorner - origin.  
"Draw straight lines connecting corners."  
aGraphicsContext displayLineFrom: origin to: topRightCorner.  
aGraphicsContext displayLineFrom: topRightCorner to: bottomRightCorner.  
aGraphicsContext displayLineFrom: bottomRightCorner to: bottomLeftCorner.  
aGraphicsContext displayLineFrom: bottomLeftCorner to: origin
```

where method *topRightCorner* calculates the top right corner on the basis of simple trigonometry

topRightCorner

```
^origin + ((self width * angle cos) @ (self width * angle sin) negated)
```

and the remaining corner finding methods are left as an exercise. To test the new class, execute the following program to display several rectangles in the active window

```
| anyRectangle |  
anyRectangle:= AnyRectangle origin: 10@80 corner: 50 @ 130 angle: 0.  
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.  
anyRectangle moveBy: (100@50).  
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.  
anyRectangle moveBy: (100@-50); rotateBy: (Float pi / 2).  
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.  
anyRectangle moveBy: (100@50); rotateBy: (Float pi / 4).  
anyRectangle displayStrokedOn: Window currentWindow graphicsContext.
```

Exercises

1. Explore class `Rectangle` and write its short description, listing its major behaviors. Comment of the number of useful behaviors that `AnyRectangle` inherits from `Rectangle`.
2. Complete `AnyRectangle` and test it. Check whether defining it as a subclass of `Rectangle` is appropriate and which of its behaviors might have to be redefined.
3. Add a method for drawing a filled rectangle. (Hint: Examine `Rectangle`.)
4. Drawing editors usually allow shrinking and stretching a geometric object. Decide how this could be done and implement the appropriate methods.

Conclusion

VisualWorks library includes a large hierarchy of number classes, mainly because hardware represents and processes different numbers differently. Another reason why there are so many number classes is that some computer applications require numbers that cannot be directly processed by hardware but are very easy to implement and provide extra programming convenience.

The most important number classes are integers and floating-point numbers. Smalltalk integers are implemented by three classes that separate the class directly implementing arithmetic in hardware from those that are added to provide unlimited integer range. Conversions from one class to another are transparent - performed automatically as needed.

Floating-point numbers are available in two implementations that differ in the number of bytes internally used to represent them and provide different accuracy and range. Conversion between high precision and ordinary floats is *not* automatic. In spite of their high precision, floating-point numbers are inherently inaccurate for decimal calculation because decimal numbers are incompatible with computer binary codes. Ignoring this principle may have serious consequences for comparison and iteration with floating-point ranges and increments. Floating-point arithmetic is also much slower than `SmallInteger` arithmetic and floating-point numbers require more memory. Floating-point numbers should thus be used only when necessary.

In addition to integer and floating-point numbers, Smalltalk also implements fractions with an integer numerator and denominator, fixed point numbers with accurate internal representation and a fixed number of decimal digits, complex numbers, and various kinds of special numbers representing values such as infinity. Of these, fractions and fixed-point numbers are the most important. They are inherently accurate but their arithmetic is very slow.

All Smalltalk number objects understand many messages implementing various kinds of arithmetic, mathematical, and conversion operations. Arithmetic is implemented on the basis of primitives and double dispatching. Primitives are messages translated into CPU code and implemented without further message sends. Double dispatching means re-sending (re-dispatching) a one-argument message to the argument, using the original receiver as the argument. Double dispatching eliminates tests because the dispatched message implies the type of the argument. Although the main use of double dispatching is in arithmetic, the principle has general validity.

In addition to calculations, numbers are also used for iteration - repetition of a block of statements a fixed number of times or over a range from a given starting value to a given end value, in specified or default fixed increments. The latter messages use blocks with internal arguments.

Blocks may have zero, one, or more internal arguments and the number of arguments is given by the type of value message used to evaluate the block. All value messages for block evaluation are defined in class `BlockClosure`. Blocks may also have internal temporary variables and their use is preferable to temporary variables declared in the surrounding context because closed blocks independent of the surrounding context are more efficient.

Profiling and timing refers to measurement of the time, message send profile, and memory required to execute a block of code. We have shown that `millisecondsToRun`: defined in class `Time` provides a simple but primitive way of timing. The Smalltalk profiler tool produces much more accurate and detailed answers.

Information hiding makes it safe to change internal implementation of an existing class. As long as the message interface does not change, changes of internal implementation don't affect existing applications that depend on the class.

When implementing a method that is an extension of an inherited behavior, we usually execute the inherited behavior and then additional code. Reference to inherited behavior (methods defined higher up in the class hierarchy) can be obtained by using identifier `super`.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

ArithmeticValue, *Date*, *Double*, *FixedPoint*, **Float**, *Fraction*, **Integer**, **Number**, LargeInteger, LargeNegativeInteger, LargePositiveInteger, MetaNumeric, Random, *Rectangle*, SmallInteger, *Time*.

Terms introduced in this chapter

block argument - argument defined and used inside a block; specified using syntax [*x* :*y*| ...]

closed block - one whose body does not depend on the external context

double dispatching - re-sending a one-argument message to the argument, with the original receiver as argument

fixed-point number - number with a fixed number of decimal digits

floating-point number - number with decimal point

integer number - number without decimal point

nested block - block within another block

message expression implemented directly by machine code and used instead of a message send

profiling - experimental analysis of program behavior with regards to its execution speed, relative frequency of message sends, and memory requirements

random number generator - an object that can produce an unlimited stream of numbers randomly distributed according to some probabilistic formula

temporary block variable - variable declared and valid inside a block; distinct from block argument

Chapter 6 - Design of applications with graphical user interfaces

Overview

In this chapter, we shift our attention temporarily to applications with graphical user interfaces (GUIs). Since our focus is on principles of user interfaces, our applications will be very simple and we will concentrate on implementation rather than design. More complicated examples will be presented later.

VisualWorks applications rely on a three-part structure GUI ↔ application model ↔ domain model. The GUI is what the user sees and interacts with, the domain model is the collection of classes modeling the objects in the problem world, and the application model is the link between the GUI and the domain model. The application model converts GUI events such as keyboard input to domain model calculations and communicates changes of values in the domain model back to GUI components. This initiates updates of the user interface. Since all application models have much common behavior, VisualWorks provides class `ApplicationModel` with the shared functionality and all applications define their application models as subclasses of `ApplicationModel`.

The operation of VisualWorks GUI components is based on the separation of the display (view), the user interaction (control), and the object responsible for the displayed data (model). This separation is called the model-view-controller or MVC paradigm.

The implementation of the model uses a special object called the value holder. Value holders encapsulate a value and keep track of their 'dependents'. When the value of a value holder changes, it automatically broadcasts a notification to all its dependents and they respond appropriately. This dependency provides a mechanism for linking the view part of GUI widgets to their models.

An important concept of the `ApplicationModel` is that it provides several 'hooks' - methods that are always executed when the application opens or closes the application. By re-defining these methods in your application model subclass, you can control the start-up and closing of your application.

6.1. Example of application development: An application selector

The main purpose of this chapter is to show how to develop an application with a graphical user interface (GUI). In this section, we will present the principles of VisualWorks applications, give the specification of a very simple problem, and outline the solution. Details of implementation will be presented in the following sections.

Principles of VisualWorks applications

Before we can start designing an application, we must understand how VisualWorks views an application. VisualWorks application architecture is based on the view that a typical application has a *graphical user interface* (GUI) that allows user interaction, a *domain model* - a collection of objects representing the entities from the problem world, and an *application model* that connects the GUI with the domain model (Figure 6.1). (VisualWorks also provides means to create *headless* applications with no user interface but in this book, we will deal only with *headfull* applications with a GUI.)

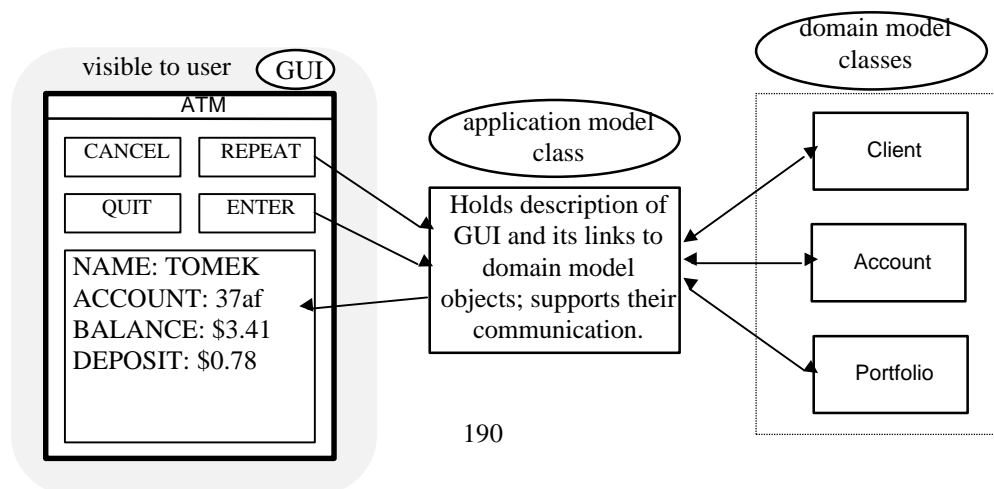


Figure 6.2. Typical VisualWorks applications consist of a graphical user interface (GUI), an application model, and a domain model. Very simple applications don't have a domain model, headless applications don't have a GUI and an application model.

Here is how an application implemented as a GUI-Application Model-Domain Model triad works:

1. User executes a message to open the application.
2. The windows open and the following sequence is repeated until the user terminates execution:
 - a. The user creates an *input event*, for example clicks a mouse button over a check box widget.
 - b. The widget sends a message to the application model.
 - c. The application model informs appropriate objects in the domain model, for example, requesting recalculation of the balance of a client's account.
 - d. Domain model objects perform the necessary processing and inform the application model of their changes.
 - e. The application model notifies the appropriate GUI components of the change.
 - f. Affected GUI components request the new values and redisplay themselves.

Application Selector Specification

We are to develop a program to allow the user to choose and run selected applications. The user interface will be as in Figure 6.1 - a window with a group of buttons labeled with the names of examples, a *Help* button, a *Run* button, and a *Quit* button. When the window initially opens, the Text View on the right instructs the user to select an application by clicking its button. Doing so displays a brief description of the selection. When the user then clicks *Run*, the selected application opens. Clicking *Help* opens a help window with general information, and *Quit* closes the window and ends execution.

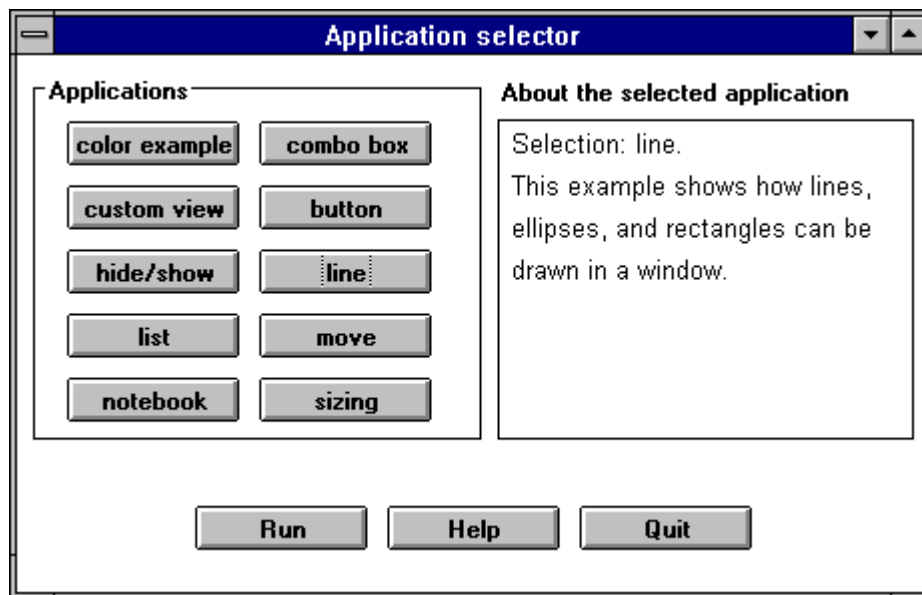


Figure 6.1. Desired user interface of *Application Selector* program. Button *assistant* has just been clicked.

Design

Our application is so simple that we don't need any domain model - the applications activated by the buttons already exist, and we don't need any data beyond the string displayed in the Text View (Figure 6.1). We will thus restrict ourselves to the GUI and the application model, a class to be called `ApplicationSelector`.

What about the GUI classes themselves? Do we have to define classes to draw the buttons and the Text View, to respond to mouse clicks, to draw the window, to redraw a piece of a window when it is uncovered after having been obscured by another window, and so on? Fortunately, we don't because all these functions are included in the library. In this example, *the only class that we will have to design is class ApplicationSelector*.

Designing a class means deciding its functionality, defining the information that it holds in its instance variables, and selecting its place in the class hierarchy. The decision as to where to put `ApplicationSelector` in the class hierarchy is simple: If we want to benefit from the built in functionality of VisualWorks application support, `ApplicationSelector` must be a subclass of `ApplicationModel`. We will explain the principles of `ApplicationModel` later but for now, we will use it as a black box. The place of `ApplicationSelector` in the class hierarchy will thus be as follows:

```
Object ()
  Model ('dependents')
    ApplicationModel ('builder')
      ApplicationSelector (???)
```

where the question marks represent the instance variables of `ApplicationSelector` that still remain to be determined.

What are the *responsibilities* of `ApplicationSelector`? It must

- open the window
- respond to example buttons by displaying explanatory text in the Text View
- know which example is currently selected so that it can be run when the user clicks the *Run* button
- respond to *Help* and *Quit* buttons
- hold information needed to display the text view and the help window.

`ApplicationSelector` thus needs an instance variable to hold a reference to the name of the currently selected example (we will call it `currentSelection`) and possibly some additional instance variables to hold information for the GUI widgets. We will see which additional variables will be needed when we learn more about the widgets. A more detailed account of the desired functionality can now be stated as follows:

- Clicking an example button sends a message to `ApplicationSelector` which stores a reference to the name of the class that will run the clicked example in variable `currentSelection`. The message then obtains the text to be displayed in the Text View, and makes sure that the Text View displays it.
- Clicking the *Run* button sends a message asking `ApplicationSelector` to open the application whose class name is stored in `currentSelection`. If no application is selected, clicking *Run* opens a notifier telling the user to select an example. (This is a reasonable extension of the original specification.)
- Clicking *Help* opens a help window with general help information.
- Clicking *Quit* asks `ApplicationSelector` to close the window and terminate.

We now have enough information to start implementing the application. If we knew more about VisualWorks GUIs, we would know which variables are required for the widgets and we could include them in our list now. As it is, we must first learn about VisualWorks' GUI.

Main lessons learned:

- VisualWorks application architecture has three components - a graphical user interface (GUI), an application model, and a domain model.
- Common GUI components are included in the Smalltalk library.
- The domain model is a collection of classes representing the problem world.
- The application model provides the link between GUI components and domain objects, and holds the description of the layout of the user interface.
- Application models are direct or indirect subclasses of the built-in class ApplicationModel.
- In very simple applications, domain information may be implemented as instance variables of the application model. In such cases, there is no need for a distinct domain model.
- An input event is an action caused by an operation such as pressing a key on the keyboard or clicking a mouse button. It activates a GUI widget and sends a message to the application model.

Exercises

1. Develop conversations for Application Selector scenarios and use them to confirm that our analysis is complete.

6.2 Implementing the user interface - the window

To create a graphical user interface, paint it with the UI Painter tool: The window being designed is a *canvas*, and the components of the interface (GUI widgets) are selected from a *palette* and 'painted' on the canvas. The following is a rather detailed description of the procedure and we recommend that you execute them as you read.

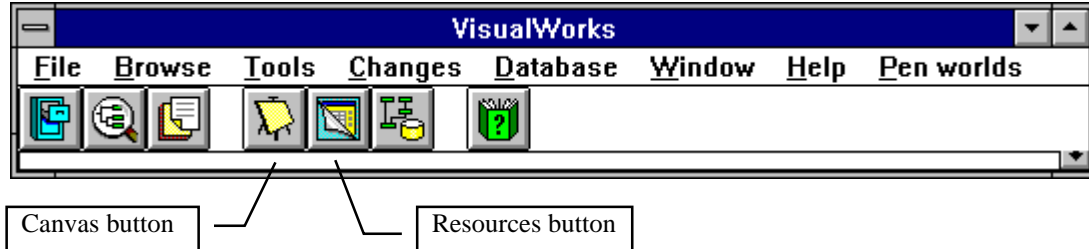


Figure 6.3. To paint a new window, click the Canvas button or use the *Tools* menu. Use the Resources button to find existing application model classes and other UI related classes.

The first step is to create a new canvas by clicking the Canvas button (Figure 6.3) or using the *Canvas* command in the *Tools* menu. VisualWorks then opens three new windows: an unlabeled canvas, a Palette of widgets, and a Canvas Tool (Figure 6.4).

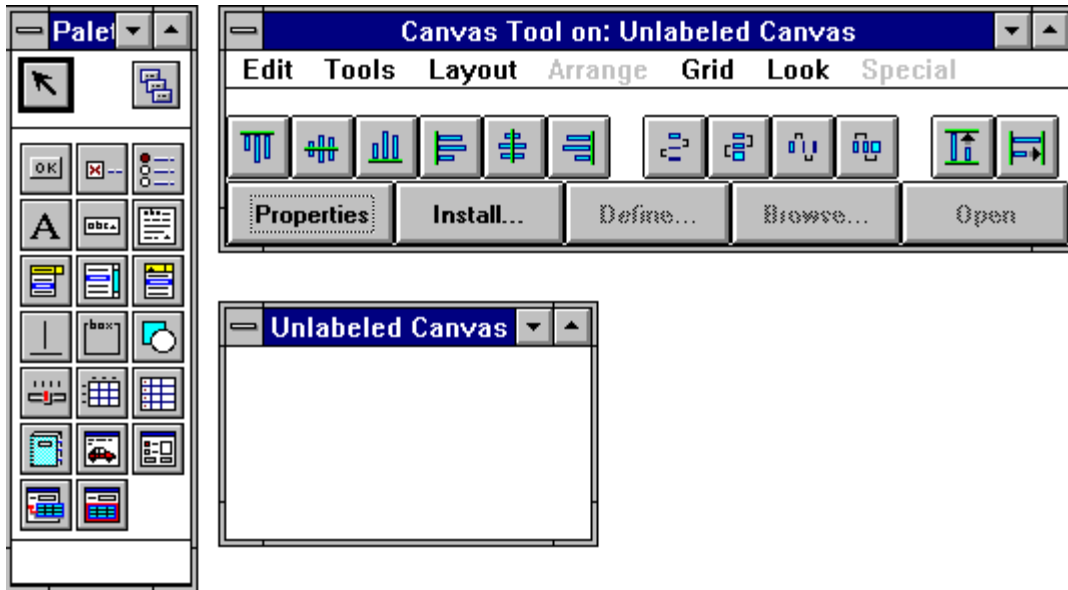


Figure 6.4. Palette with GUI widgets (left), Canvas Tool (top), and unlabeled canvas.

Installing the canvas on an application class

After creating an empty canvas, *install* it on an application model class by clicking *Install* in the Canvas Tool. (Like most other actions, this can be also be done from the <operate> menu of the Canvas itself.) VisualWorks will then lead you through a series of dialog windows to specify the name of the application class, the name of its category, the type of its interface, and the name of the class method that will hold the description . The first of these windows is in Figure 6.5.

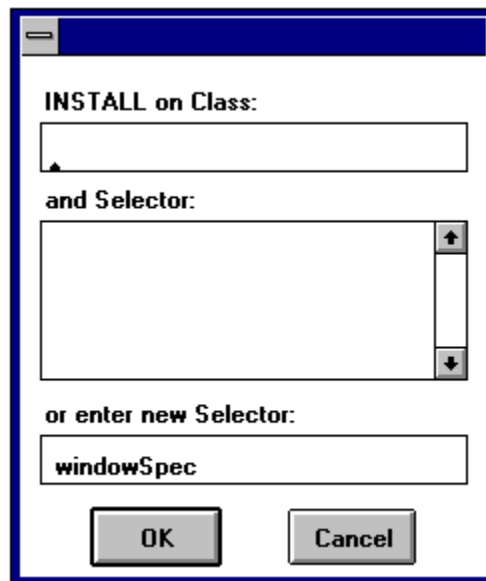


Figure 6.5. The first window after clicking *Install*.

Type the name of your application model on the first line as in Figure 6.6. If the class does not yet exist, the UI Painter will create it. Next, use the bottom line of this window to enter the name of the class method that will hold the window specification - the description of the user interface. The recommended

name for the specification method is `windowSpec` and this name is already displayed at the bottom of the window. Use this name unless your application requires several windows which must then be stored as separate specifications under different names. The advantage of using `windowSpec` is that it allows you to open and run your application simply by sending the message `open`, as in

`ApplicationSelector open`

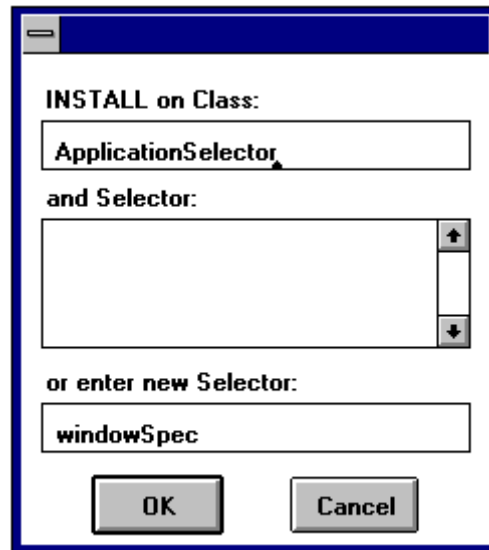


Figure 6.6. Provide the name of the application model class and Selector of the specification method.

After entering the selector of the window specification method, click *OK*. If the application model class does not yet exist, the UI Painter now opens another window to get more information (Figure 6.7). Much of this window is already filled-in. The name of the application model class is on the top line and the name of the category in which the class will be stored is on the second line. The suggested category is `UIApplications-New` but we will use the name `Tests` instead. If the category does not exist, the painter will create it for you.

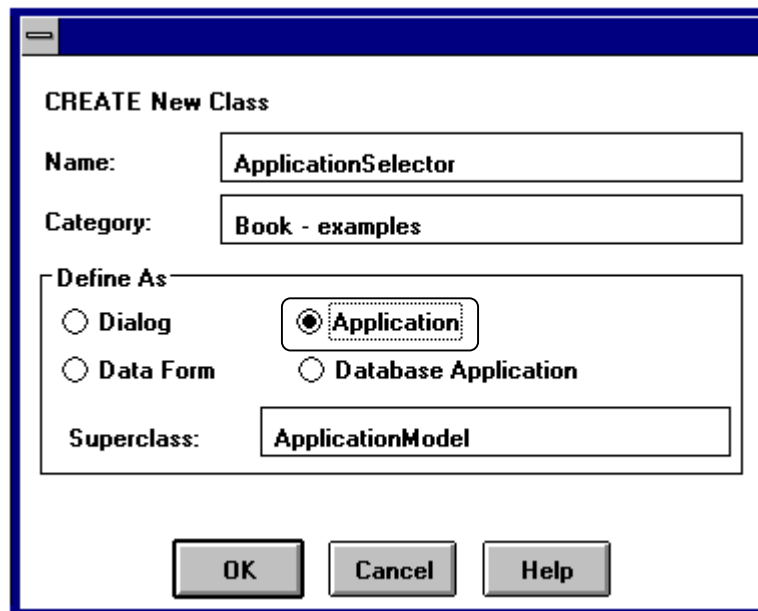


Figure 6.7. Specifying category, superclass, and type of user interface application model. We only had to click the *Application* button and change the name of the category.

The next step is to select the type of the window in the box labeled *Define As*. This determines the behavior of the window. In most cases, the desired type of interface is *Application* and we thus select the *Application* radio button. (We will explain what a *Dialog* is and how to create it later.)

Finally, we must choose the *superclass* of our application model. The window suggests *ApplicationModel* and this is usually the appropriate choice. In some cases, you may have previously created a useful application model for a related application and you may want to reuse it. If so, replace *ApplicationModel* with the name of the superclass. Note, however, that you must have *ApplicationModel* in the superclass chain or you won't be able to benefit from all features of VisualWorks GUI classes. In our case, we leave *ApplicationModel* as superclass name. After this, click *OK* and VisualWorks will create a definition of class *ApplicationSelector* with the specified superclass, and put it in the specified category. It will also create the class method *windowSpec* containing the description of the interface in its present state. You can find it on the class side of the browser and its present structure is as follows:

windowSpec

```
"UIPainter new openOnClass: self andSelector: #windowSpec"
```

```
<resource: #canvas>  
^#(#FullSpec  
  #window:  
    #(#WindowSpec  
      #label: 'Unlabeled Canvas'  
      #bounds: #(#Rectangle 300 200 500 400) )  
    #component:  
      #(#SpecCollection  
        #collection: #() ) )
```

As you can see, the method begins with `<resource: #canvas>` which makes it special and the compiler treats it differently than regular methods. The method contains a description of the window with its label and, size and position, and when you add new widgets and re-install the window, the spec method will be updated to capture the new layout. Remember that when you change the canvas (for example by adding new widgets, changing the label, or changing the background color) you must re-install the canvas. Otherwise your changes will only be painted on the screen but not captured in the library.

The class is now compiled and the UI specification saved and you can run the application, either by executing

```
ApplicationSelector open
```

in a Workspace or by clicking *Open* in the Canvas Tool. This will open an empty window looking just like the canvas that we have created.

Defining canvas properties

After installing the raw canvas, we will now define its properties. Click *Properties* in the Canvas Tool and you will get the window in Figure 6.8. Since nothing but the canvas is selected, the Properties Tool is open on canvas properties and we will redefine its label to Application Selector. Type Application Selector on the *Label* line and click *Apply and Close*. The *Properties* window closes and the canvas displays with the new label. Note that if you now opened the application, the label would be unchanged because we have not re-installed the changed canvas. *Install* again, open the application, and check that the label of the window has changed.

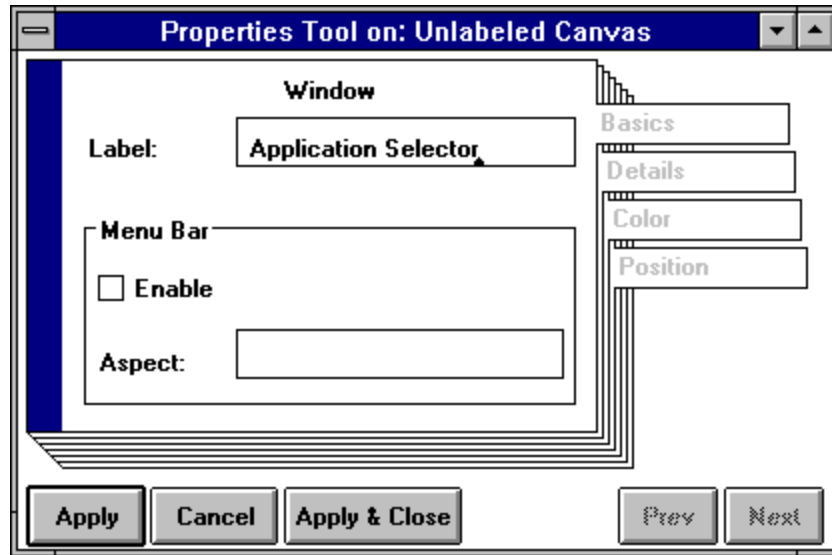


Figure 6.8. Defining a new label for the window.

We could define and install additional properties of the window such as its background color, but we will now proceed to the widgets. If you must interrupt your work at this point, close the interface and save the Smalltalk image. When you return, you can reopen the canvas editor either by

- clicking the *Resources* button (Figure 6.3) and locating your application model class in the Resource Finder window (Figure 6.9) or
- by opening the browser on your spec method windowSpec method and executing the Smalltalk statement in the comment at the top of the definition (see listing above).



Figure 6.9. You can access an application with the Resource Finder. *Edit* opens the canvas editor, *Start* opens the application on the selected specification, *Browse* opens a browser on the application model class.

Main lessons learned:

- To create a user interface and its application model
 - Open a canvas with a Palette and a Canvas Tool from the launcher.
 - Install the canvas on an application model class using the *Install* button. Complete the dialog windows, specifying the name of the method holding window specification, the name of the application model class and its category, and the type of the window (*Application*). The class will be automatically defined if it does not yet exist. To open the installed canvas, click *Open* in the Canvas Tool or send *open* to the application model class.
 - Define window properties using *Properties* from the Canvas Tool and re-install.
- If you change the user interface, re-install the window. Failure to re-install a changed GUI is one of the most common causes of a strange looking or misbehaving interface.
- Changes to application model code do not require re-installation.
- Executing the comment at the beginning of the canvas specification method (typically *windowSpec*) opens the Canvas, Palette, and Canvas Tool on the interface.

Exercises

1. Follow the procedure described in this section and create the canvas. Examine the *windowSpec* method.
2. Use the Properties Tool to change the background color of the window to light yellow. Re-install before opening and check the new *windowSpec*.
3. Most VisualWorks tools are implemented with the user interface painter. Test this by checking the *windowSpec* method of class *Browser* and opening it with *open*.

6.3 Painting widgets and defining their properties

After creating the canvas, we will now paint the widgets and define their properties. To paint a widget on the canvas, click it in the palette and drop it on the canvas, clicking the select button when the widget is positioned where you want it. Initially, you will not be able to identify palette buttons but when you click a palette button, its name is displayed at the bottom of the palette as in Figure 6.10. The Sticky Selection button allows you to make several copies of the selected component; click it on to start and click it off to disable the sticky mode.

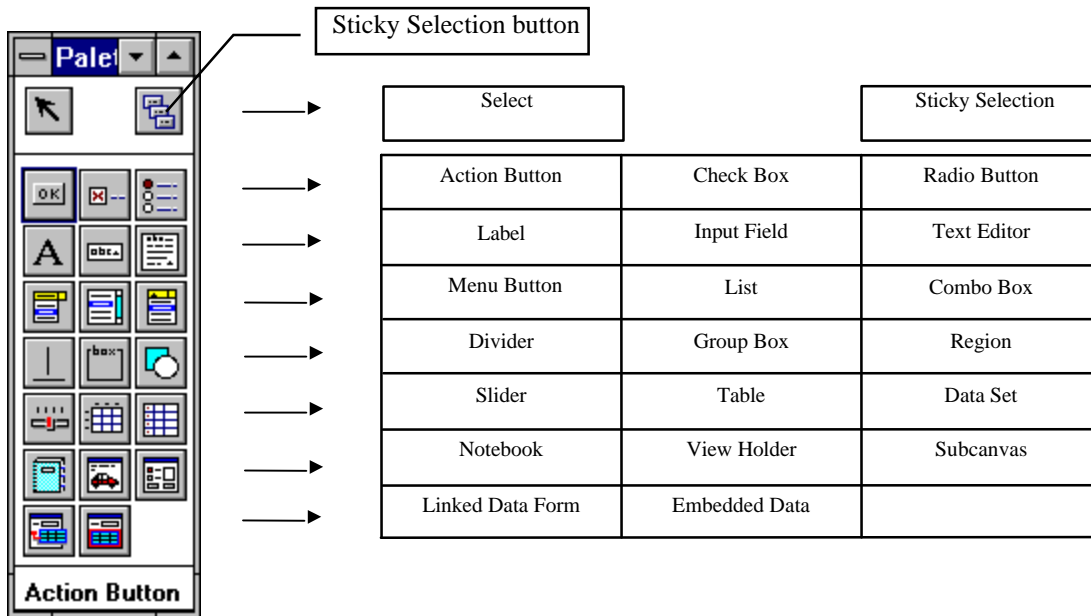


Figure 6.10. UI Palette and its buttons.

We will now paint the Action Buttons required in our interface. Click the Action Button button (!), move the mouse cursor over the canvas, and click. When an Action Button appears (Figure 6.11), click again to drop it in place. If you don't like the button's position or size, move it or reshape it by dragging its body or its *handles* - the small rectangles in the corners - while pressing the <select> mouse button.

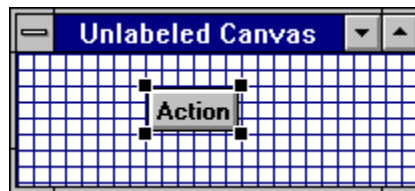


Figure 6.11. Widget handles show that the widget is selected. You can now move it, reshape it, define its properties, or delete it (command *cut* in the <operate> menu).

Position the first Action Button as in Figure 6.12 and proceed to define its properties.

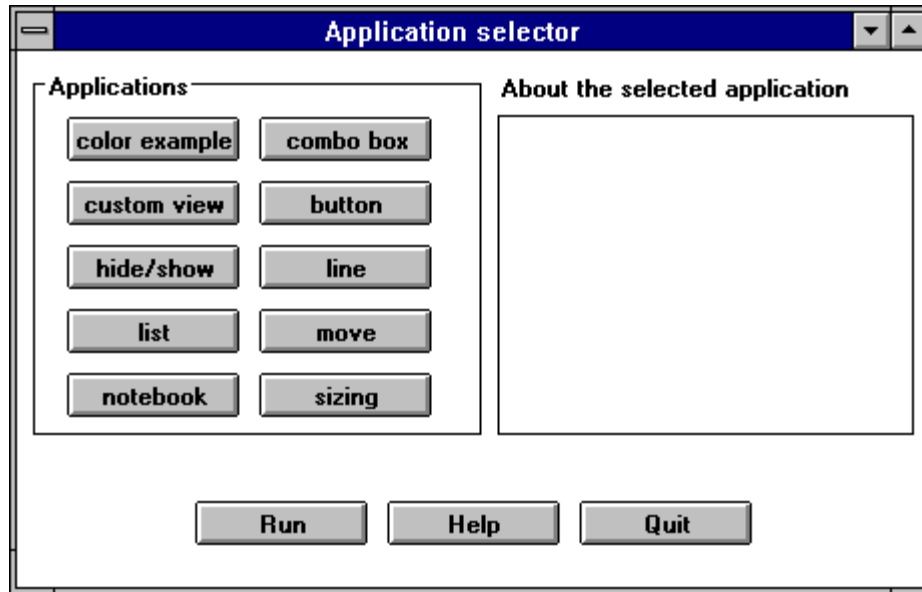


Figure 6.12. Desired layout.

To define widget properties, select the widget in the canvas and click *Properties* in the Canvas Tool. The Properties Tool window opens (Figure 6.13) and you can now define the button's *Label*, its *Action*, and other parameters.

The *label* is the text displayed inside the button - *color example* in this case - and *Action* is the name of the message that the button sends to its application model when you click it in the running application. You can use any legal identifier for the *Action* method, but the name will be easier to remember if it is similar to the label; we called it *color*. VisualWorks automatically adds the # sign in front of it, making it a Symbol, a special kind of string. We have now defined all the properties that we need. Click *Apply* at the bottom of the Properties Tool if you want to proceed and define the properties of another widget. Clicking *Apply & Close* applies the properties and closes the Properties Tool. The button in the canvas now shows the new label and you will probably have to reshape it because the button is too narrow. Note that you can choose a variety of other properties such as background and foreground colors on other pages of the Properties Tool.

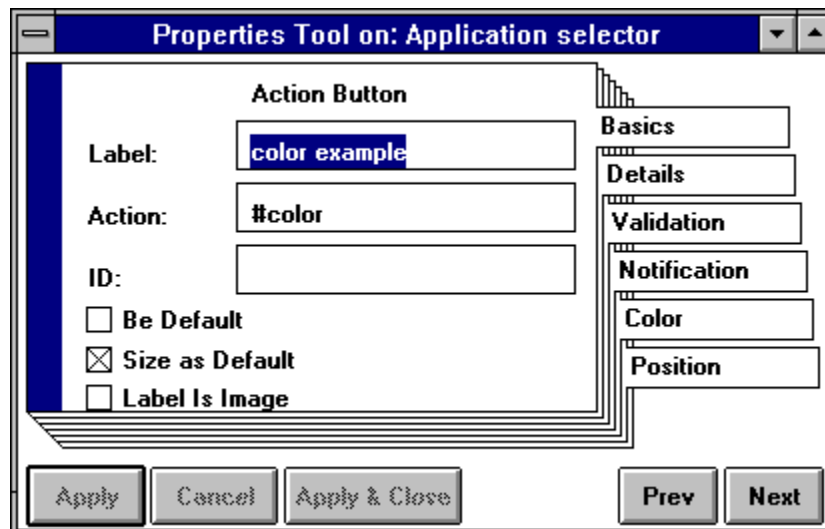


Figure 6.13. Properties window on an Action Button.

We leave it to you to create and position the remaining buttons and define their properties. One way to do this is to *copy* and *paste* from the canvas <operate> menu. With this method, the buttons will be copied with all their properties such as size, label, and action, and you will have to redefine them. Having so many buttons to paint, you might want to create all buttons in the left column first, select the whole column, *copy* it, and *paste* it to create the right column. To select a group of items, hold the <select> button down and move the mouse cursor to draw a rectangle around the widgets to be selected. When you release the button, the widgets' handles will be displayed and you can *copy* them.

Now that you have all the Action Buttons, you may want to change their widths, heights, or alignments using the alignment buttons in the Canvas Tool (Figure 6.14). The six buttons on the left are for alignment of a group of widgets along the top of the first selected widget, its center, bottom, and so on; the middle four buttons produce equal spacing between widgets, and the two buttons on the right are for equalizing widget width and height. Select the widget that you want to use as the template and then the remaining widgets by clicking the left mouse button while holding <shift> down. Then click the desired alignment button.

Install the canvas and click *Open*. The window opens with all the buttons, but the buttons will not do anything because we have not defined their action methods.

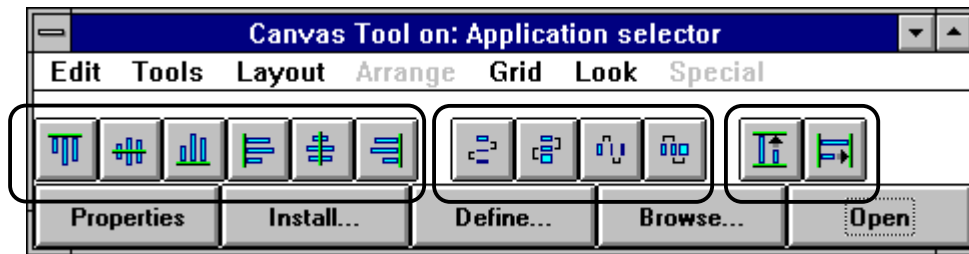


Figure 6.14. Canvas Tool buttons for automatic widget alignment.

As the next step, we will add the *Text Editor* widget for displaying the help text. Click the corresponding button in the palette, position the widget on the canvas, shape it to the desired size, open the *Properties Tool* window, and specify description as the name of its *Aspect*. The *Aspect* of a Text Editor is the name of the method that supplies the displayed text, and the instance variable containing a value holder with the text. (More on value holders in a moment.)

After defining and applying the Text View's *Aspect*, open the *Details* property page, click both scroll bars off, and the *Read Only* property on (Figure 6.15). This will remove the scroll bars from the Text Editor (our text will be short and scrolling is unnecessary), and make the text read-only, preventing the user from changing the text. Click *Apply and Close*, and *Install* the canvas again.

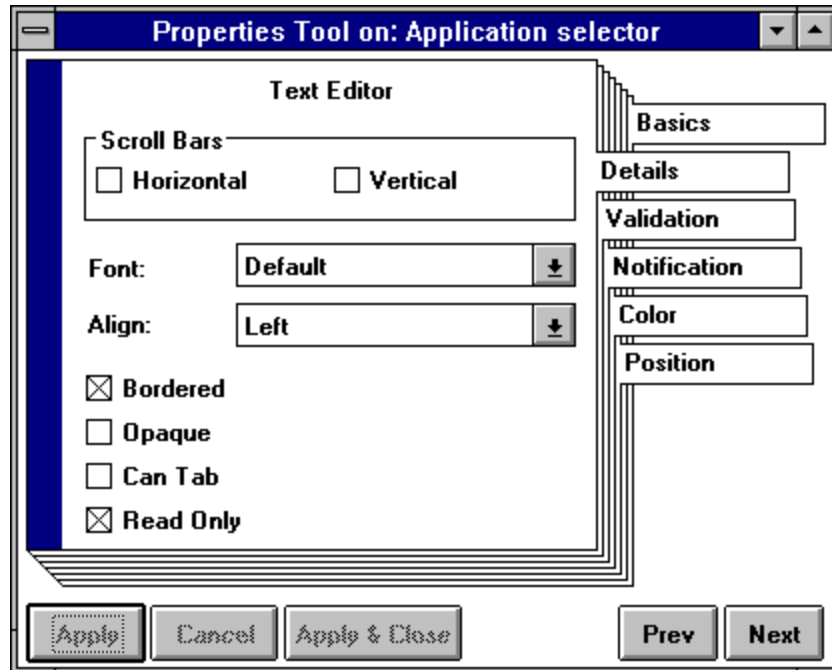


Figure 6.15. *Details* page of Text Editor's properties.

The GUI is now completely installed and you can try to run it by clicking *Open* in the Canvas Tool. This time the application will *not* open and we will get the Exception window in Figure 6.16. It says that *binding #description* was not found. This means that when the *builder* object that builds the window from *windowSpec* tried to construct the window, it sent message *description* to get the text for the Text Editor - but we have not defined this method yet. Before we can open the interface, we must thus define all aspect methods. This topic is covered in the next section.

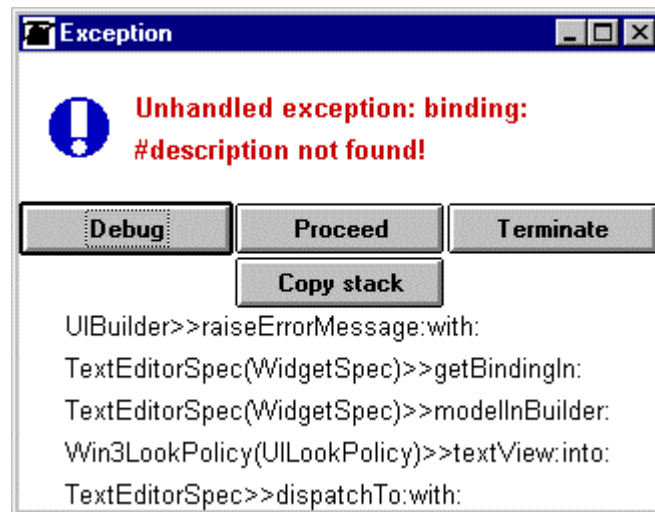


Figure 6.16. The window builder could not establish the bindings between a widget and its value holder.

Main lessons learned:

- To add widgets to an interface, import them from the palette or copy and paste widgets already on the canvas.
- When a widget is selected in a canvas, its handles are shown. Handles can be used to change the widget's shape. A selected widget can be moved around, edited, deleted, or copied.
- To align or resize a group of widgets, select the widgets and use alignment buttons in the Canvas Tool.
- To define widget properties, use the Properties Tool.
- Action Buttons have an *Action* property, most other widgets have an *Aspect* property.
- *Action* is the name of the method sent to the application model when the user clicks the button.
- *Aspect* is the name of a value holder holding a widget's value, and the name of its accessing method. The *Aspect* of a Text Editor is the message that returns the value holder with the text, and the name of the variable holding it.
- When choosing selectors for *Action* and *Aspect* methods, use names that match the label of the widget or its purpose.
- Before you can open an application, you must define *Aspect* variables and methods so that the user interface builder can establish and use the bindings between the widgets and their value holders.

Exercises

1. Follow the steps listed in this section and create the widgets of Application Selector. When you are finished, read the `windowSpec` method and explain how it changed from Section 6.2.
2. Change the color of the text of the *Run* button to green and the color of the text of the *Help* button to red.
3. The box around example buttons and the label Applications in Figure 6.1 is a Group Box widget. Add the Group Box according to the desired layout. Change the color of its text label and frame to red, and the background to light green if possible. (Hint: Use on-line help if necessary.)

6.4 Defining *Action* and *Aspect* methods

To make our buttons functional and to make it possible to display the Text Editor view, we must now define their *Action* and *Aspect* methods.

The first step in creating *Action* and *Aspect* methods is easy because VisualWorks can define their skeletons (*stubs*) for you. To do this, select all widgets in the canvas for which you want to create stubs, and click *Define* in the Canvas Tool. This will open a window (Figure 6.17) listing the *Action* or *Aspect* names of all selected widgets. Click *OK*, VisualWorks creates the stubs, and closes the window. You can now open a browser on the application model and browse the automatically defined stub methods, or run the application and click the buttons because the messages that the buttons send now exist. Of course, clicking a button will not do anything because the messages that the buttons send don't have any meaningful body because we have not specified what they should do. To get the application to work, we must now redefine the bodies of all the *Action* and *Aspect* method stubs.

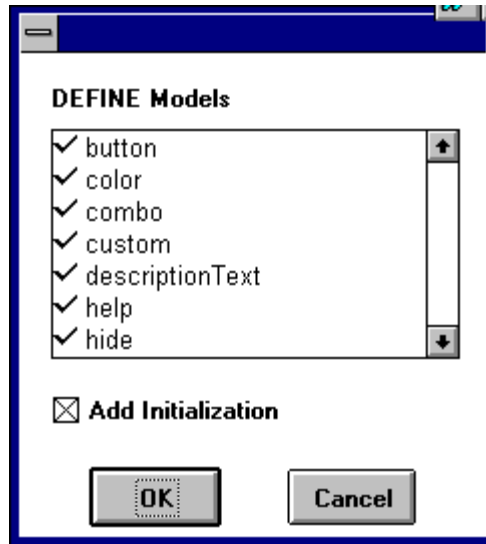


Figure 6.17. Automatic definition of stubs of widget methods. When the *Add Initialization* check box is on, VisualWorks includes lazy initialization in the definition of all selected *Aspect* methods.

Although we could define all *Action* and *Aspect* methods in the System Browser, we will now use a smaller browser restricted to our application model class and its hierarchy. To open it, deselect all widgets in the canvas by clicking outside their perimeters, and click the *Browser* command in the Canvas Tool. This opens a *hierarchy browser* as in Figure 6.18. (When a widget is selected in the canvas, the browser opens only on methods relevant to this widget.)

Note that we now have an instance variable called *description Text*, the *Aspect* of the Text Editor; it was created by the *Define* action. For completeness, add instance variable *currentSelection* and recompile the class with *accept*.

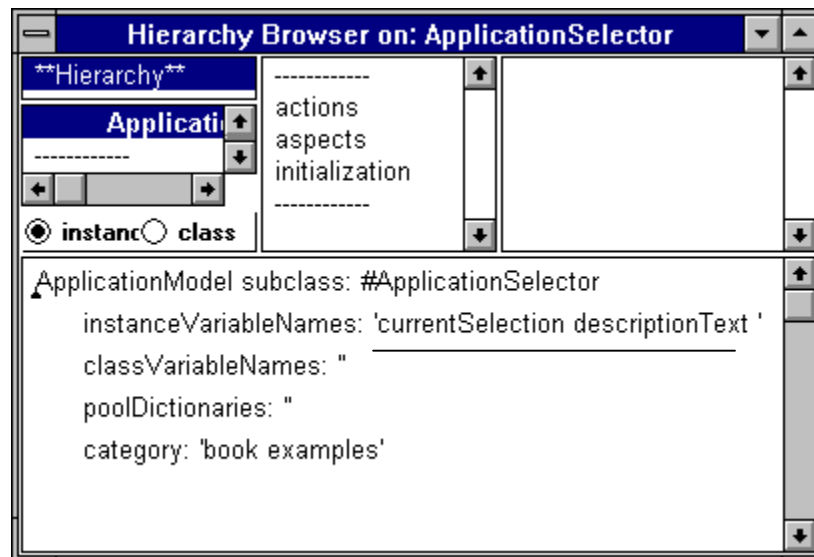


Figure 6.18. Hierarchy browser of class *ApplicationSelector*.

Our next task is to fill in ('flesh out') the bodies of the stub methods stored in instance protocols *actions* and *aspects*. Protocol *actions* holds *Action* methods for Action Buttons, protocol *aspects* holds *Aspect* methods, in this case method description for the Text Editor. We will begin with the button method in the *actions* protocol.

Click actions and button in the Browser to display the code in Figure 6.19 which shows that the body of the method does not do anything. Its only use is that if we run the application and click the button, the application will not crash because the message sent by the button *exists*.

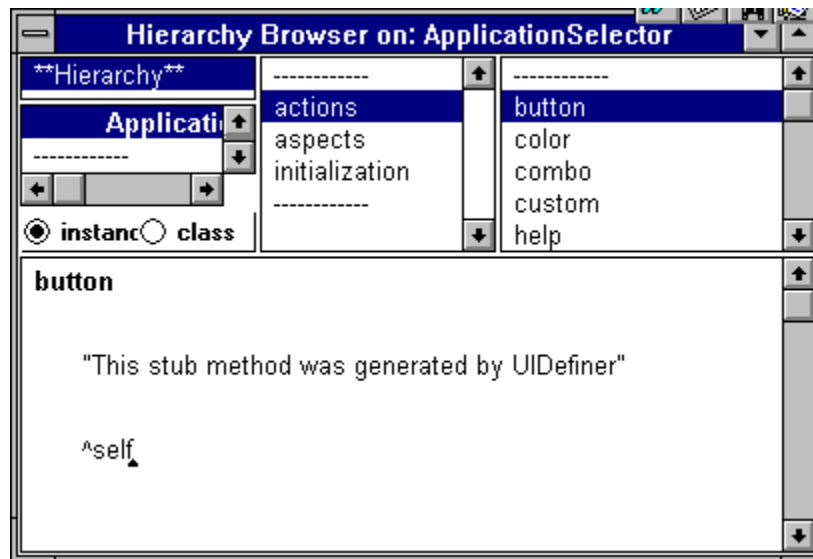


Figure 6.19. Stub definition of button created by the *Define* command.

What do we want the button method do? According to our design specification, the method should

- store a reference to the name of the example class ButtonExample in variable currentSelection,
- supply the description text, and
- tell the Text Editor view to display it.

To tell the Text Editor to display the text, we must ask variable description to change its value by sending it the value: message with the text as its argument. The desired definition is as follows:

button

"Make ButtonExample the current example and display its short description in the text view."

currentSelection := ButtonExample.

description value:

'Current selection: button.'

This example illustrates the three types of buttons available in VisualWorks:

Action Button sends a message.

Check Box Button sets its value to true/false.

Radio Button turns itself on and other radio buttons in the same group off.'

After entering and accepting this definition, open the application and test that the *button* button now works. Note that we did not have to reinstall the canvas because we did not change the GUI - we only changed the code of the class.

We are leaving the remaining Action Button *Action* methods to you. You only need to know that the names of the example classes are ColorExample, ComboBoxExample, CustomViewExample, HideExample, LineExample, List1Example, MoveExample, NotebookExample, and SizeExample. (Test the applications by sending them open to decide on the description text.) After entering and accepting these definitions, you can run the application and all example buttons will now work.

Before proceeding to the remaining methods, we will now make a small change to our canvas to show how easy it is to change your user interface – we will define the size of the window as *fixed* so that the user cannot change it. This will be useful because our Text Editor view cannot be scrolled and if the user made the window smaller, the text might not be readable. To make the size of the window fixed, click

Layout in the Canvas Tool and select *fixed size*. Don't forget to *Install* the new layout and test it! If you now open the application, you won't be able to change the size of the window.

Main lessons learned:

- When you select widgets in the canvas and execute *Define*, Smalltalk creates stub definitions of their *Action* and *Aspect* methods of and defines their instance variables. The names are determined by the properties of the widgets.
- Stub definitions usually don't do anything but their existence makes it possible to open the application and click its widgets. It also saves you from having to remember the names of aspects and actions when you want to write their complete definitions.
- After creating the stubs, you usually have to edit them so that they perform their intended tasks.
- Changing the user interface requires only editing and re-installing it.

Exercises

1. Implement the material covered in this section.

6.5 The remaining *Action* methods

In this section, we will define *Actions* for *Help*, *Run*, and *Quit*, the remaining Action Buttons.

Help

The *Help* button opens a window with general help. We could create a special window for help using the UI Painter, but it is easier to use the built-in class *SimpleHelp* which opens a help window with an *OK* button at the bottom. To find how to use it, we can either read its definition in the browser, or we can look for references to *SimpleHelp* in the library and find how existing code uses it. Using the second approach, we find *SimpleHelp* in the browser, select command *class refers* in the <operate> menu of the class view, and examine several examples. We find that to use *SimpleHelp*, we must

1. send it the class message *helpString: aString* to create an instance with the help text *aString*
2. send *open* to open the window with this text¹.

The whole definition of the help message which is sent by the *Help* button is thus as follows:

help

"Open help window with general text on the whole application."

(*SimpleHelp* *helpString:*

'Each of the buttons on the left provides access to an example application.

Clicking one of these buttons, displays information about the selection in the Text View.

If you then click Run, the application will open.') *open*

Enter and *accept* this definition and test that the *Help* button works. Note again that changing the definition of an *Action* method does not require re-installing the canvas – it's not a user interface change. Test the *Help* button now.

Run

Clicking *Run* will open the currently selected application. Since all our example applications use *windowSpec*, they can be opened by *open*. We stored a reference to the name of the class of the selected application in instance variable *currentSelection* and the definition of *run* is thus as follows:

run

¹ *SimpleHelp* is a subclass of *ApplicationModel* with its window specification in *windowSpec*.

"Open the currently selected example application."
currentSelection open

This definition works fine if the user previously clicked an example button but if the user has not done so, the program will crash because currentSelection is nil, and nil does not understand message open. To deal with this possibility, run must check the value of currentSelection and send open only if it is not nil:

run

"If an example application is selected, open it."
currentSelection isNil
ifTrue: [Dialog warn: 'You must select an application first.']
ifFalse: [currentSelection open]

Test that everything works so far.

Quit

The *Quit* button should close the window and terminate the application and this can be done, for example, by sending message closeRequest to the application model. The definition of quit is thus

quit

"Close the window and terminate the application."
self closeRequest

Main lessons learned:

- To find how to use a class or a method, read its class or method comment or check existing uses.
- To run an application whose model is a subclass of ApplicationModel and whose interface is stored in class method windowSpec, send the class message open.
- To close an application and its window, send closeRequest to the application model.

Exercises

1. Implement and test the material covered in this section.
2. Implement *Help* using Dialog warn: instead of SimpleHelp.

6.6 Text Editor widget

The only unfinished part of our application is the Text Editor and we will now implement it starting from the stub definition of its *Aspect* method description Text. Its current form, automatically generated by *Define*, is as follows:

description Text

"This method was generated by UIDefiner. *Any edits made here may be lost whenever methods are automatically defined.* The initialization provided below may have been preempted by an initialize method."

```
^description isNil  
ifTrue: [description := String new asValue]  
ifFalse: [description Text]
```

Before we explain this code and modify it for our needs, note that the comment says that if you redefine the method in the Browser and then use *Define* on this aspect, your definition will be replaced by this default version. So if you change the definition, don't use the *Define* button on it again or you will lose your work. Now back to the definition.

The code first checks whether the value of description is nil. Why? The reason is that when the application first opens, description is nil and when the builder sends description to obtain the text as it is

building the window, the process would fail because the builder could not build the text view from nil. The method thus assigns

```
description := String new asValue
```

which returns an empty String, packaged as a ValueHolder object (to be explained shortly). The window extracts the empty string from the ValueHolder and displays it - no text in the description view. This technique of not initializing a variable until initialization is really needed is called *lazy initialization*. It is useful especially if initialization is time consuming, if it is not required immediately, and if the value may not be required during each execution of the application. Everything now works and you can check that when you execute

ApplicationSelector open

everything works as desired. The only blemish is that when the application opens, the Text View does not display anything - and we would like it to display the message 'Please select an example application.'. To correct this problem, we must store this string in description *before* the window opens and this can be done by defining instance method initialize which is automatically executed as a part of the application opening process. (We will explain how this happens in a moment.) Since the value of description must be a ValueHolder, the definition of initialize in class ApplicationSelector will be as follows:

initialize

```
"Assign initial text to Text Editor via its associated Aspect value holder."  
description := 'Please select an example application.' asValue
```

where message asValue converts the string into a value holder holding the string.

The program now works just as desired but the body of the description method that was automatically created by *Define* is now partially redundant: Variable description is now guaranteed to have a value when it is first requested by the builder (message initialize is sent before the builder starts creating the window), lazy initialization is no longer required, and the ifTrue: part of

description

```
^description isNil  
  ifTrue: [String new asValue]  
  ifFalse: [description]
```

will thus never be executed. All this method will ever do is return description Text. Although the method still works, it is nicer to delete the unnecessary part:

description

```
^description
```

Before closing this section, we must mention one additional detail about Text View widgets. As we have seen, a Text View has an *Aspect* variable to hold the value holder with its text and this variable is declared in the application model class. However, each Text View distinguishes *two* text objects. One is the text displayed in the widget, the other is the text last accepted with the *accept* command or initially assigned to the widget. Text View holds the currently displayed text in a variable associated with the widget itself, and the accepted in the *Aspect* variable in the application model (Figure 6.20). This makes it possible to restore a Text View to its original contents with *cancel* when the user changes but does not *accept* its contents. As programmers of the application, Our interest is usually restricted to the aspect variable in the application model and we can ignore the instance variable of the widget.

Text displayed in window

Some text displayed on the screen - we have just changed it but we have not clicked *accept* yet.

Last *accepted* text, possibly different from text now displayed

This was the text on the screen when we last clicked *accept* in the widget and before we made the changes on the left..



Figure 6.20. The text displayed in a Text Editor is stored in the widget's variable, the accepted text is stored in the *Aspect* variable in the application model.

Main lessons learned:

- Lazy initialization means leaving the initialization of a variable until the variable is needed.
- The stub created for *Aspect* methods by the *Define* command implements lazy initialization.
- To display a string in a Text Editor when the window first opens, initialize its *Aspect* variable to the desired string converted to a *ValueHolder* using message *asValue*. Perform this initialization in application model instance method *initialize*.
- The *Aspect* variable of a Text Editor must be a *ValueHolder* on a *String*. It holds the accepted text.
- The Text Editor widget holds the currently displayed value in its own instance variable. The accepted value is held in the application model.

Exercises

1. Implement the material covered in this section.
2. We used *Define* to create the definition of description and then edited it to remove lazy initialization. If you don't intend to use lazy initialization, click off the check box *Add initialization* in the *Define* window and leave the body of the method empty. Test this feature.
3. Find how message *asValue* works.

6.7 Value holders, models, and dependents

GUI widgets fall into several categories:

- Passive widgets that organize the window but don't respond to user input or changing values of domain objects. Labels and grouping boxes belong into this category.
- Active widgets that respond to user input or changes of domain objects. This category can be further subdivided in widgets that
 - invoke actions – such as action buttons
 - gather input – such as text editors
 - display information – such as lists.

Widgets that display or gather information need an object to hold the displayed information and this object is generally some kind of a value holder. We have already mentioned that a *ValueHolder* is a capsule containing a value (any object) and providing several accessing messages and a built-in mechanism for communication with 'dependents'. We will now explain the value-related messages and the concept of dependency.

ValueHolder message interface

To put an object in a *ValueHolder*, send it the conversion message *asValue* such as

'A piece of text' asValue

which creates a ValueHolder containing the string 'A piece of text'. As another example,

x := 3.14 asValue

creates a ValueHolder containing the Float object 3.14.

Creating value holders with asValue is most common but ValueHolder also implements several specialized creation messages including newBoolean, newFraction, newString which create value holders on True, 0.0, and an empty string. You can also use the class message with: anObject which creates a ValueHolder on anObject as in

ValueHolder with: String new

The accessing protocol of ValueHolder includes value, setValue: and value:. Message value returns the object in the value holder. As an example, sending value to the variable used above, as in

x value

returns 3.14. As another example, to obtain the value of variable description in our Text Editor, execute

description value

To *change* the of a value holder *and* to notify its dependents (explained below), send value: as in

description value: 'New help text'

Remember that if you want to preserve the magic of widget value holders, you must not change description value by

description := 'new help text'

because this would change the nature of description (Figure 6.21) from a ValueHolder holding a String to a String. This would destroy the dependency between the value and its widget, and the part of the GUI that depends on description being a ValueHolder would now stop working. Unfortunately, assigning a value to a variable that holds a ValueHolder instead of using value: is a very common mistake!

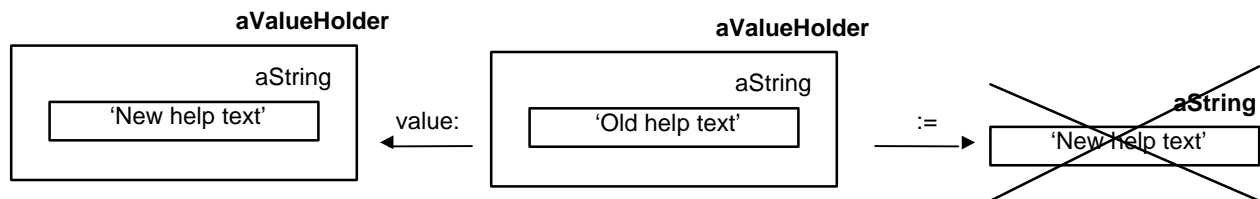
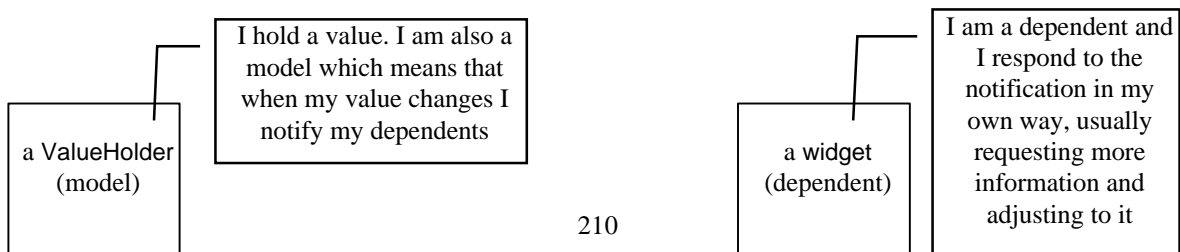


Figure 6.21. Proper and improper ways of changing the value of a ValueHolder. Center: original state. Left (correct): result of description value: 'New help text', right (incorrect): result of description := 'New help text'.

Value holders are most commonly used as *models* of widgets, as objects that hold the value of a widget and enforce *dependency* of the widget on any changes of this value caused by the application (Figure 6.22). Note, however, that the use of the principle of a model-dependent object is not restricted to widgets.



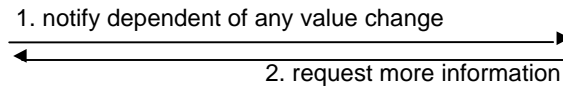


Figure 6.22. The two sides of the model \leftrightarrow dependent relationship in GUI widgets.

The dependency relationship between a value holder and a widget is established by UIBuilder when it constructs the user interface. VisualWorks refers to this link as a *binding* and we will now explain in more detail how the mechanism works. We will use the example of the Text Editor in our Application Selector.

When the builder builds the user interface from the `windowSpec` method, it associates a binding between the aspect variable description and a `ValueHolder` assigned to the Text Editor widget as its model. Once the binding is established and the window opens, it is used as follows: When the applications sends `value:` to the `ValueHolder` (Figure 6.23) as in

description value: 'New help text'

the `ValueHolder` changes its value and notifies its single dependent, the Text Editor, that it has changed. It does so by sending it the `update: #value` message. The definition of `update:` in the Text Editor widget (class `ComposedTextView`) responds by sending `value` back to its model, the `ValueHolder`. When it gets the result – the new text – it redisplay itself.

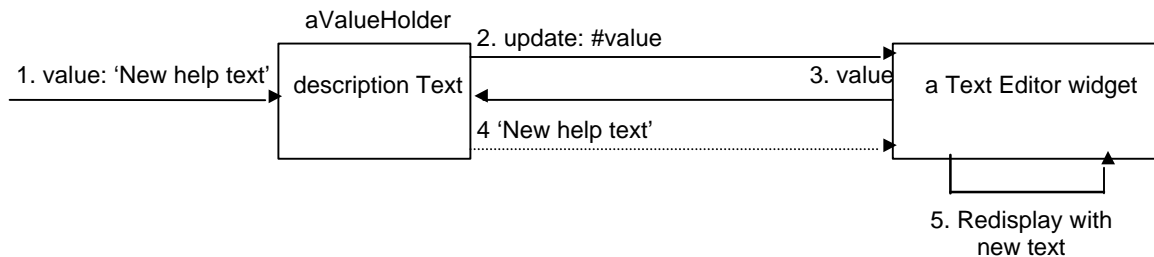


Figure 6.23. The chain of events resulting from sending `value: 'New help text'` to description. Full lines represent message sends, interrupted lines indicate returned objects.

To keep track of its value and its dependents, `ValueHolder` has two instance variables called `value` (holds stored value) and `dependents` (holds pointers to all dependents). The following definitions show exactly how `ValueHolder` works:

value

`^value`

value: newValue

"Set the currently stored value, and *notify dependents*. Declared in superclass `ValueModel`."

`self setValue: newValue.`

`self changed: #value` "This is where the notification occurs."

where

setValue: aValue

" Just change the value *without notifying dependents* of a change. "

`value := aValue`

changes the value but does not notify dependence. Dependency is defined by message `changed:` which triggers the broadcast of `update:` to the dependents. Method `changed:` is declared in class `Object` essentially as follows:

changed: anAspectSymbol

"The receiver changed. The change is denoted by anAspectSymbol. Inform all dependents."
self myDependents update: anAspectSymbol

We have not shown the additional detail involving the changed: message in Figure 6.23.

The ValueHolder thus triggers change notification via the update: message but for this mechanism to work, each dependent must understand the update: message; the way in which it defines update: then determines how it responds. We have seen that the definition of update: for the Text Editor widget, for example, asks the model for the new value of the string and redisplay itself with the new string. To take care of cases in which a dependent does not care about the update: message, the basic definition in class Object does not do anything. Every object thus understands update:.

We can now trace the full meaning of the definition of the button method from the previous section:

button

currentSelection := ButtonExample.
description value: 'the text for the help comes here'

After assigning a new value to currentSelection, button sends value: its ValueHolder, which then notifies its dependent (the Text View) via changed and update. The Text Editor view asks its value holder description for its value, gets the new text, and displays it.

Creating widget - value holder bindings

We have already mentioned that the bindings between widgets and their value holders are established during the opening of the application. The process is as follows (Figure 6.24): When you send open to your application model, it creates a new instance of the application model and a user interface builder, an instance of UIBuilder. The builder object then builds the window in your computer's memory from windowSpec, creating bindings between widgets and their value holders. This is achieved by sending *Aspect* messages to the application model. In our example, UIBuilder sends description to ApplicationSelector which returns a ValueHolder with the initialized string. The UIBuilder then associates this value holder object with the Text Editor and makes the widget a *dependent* of the aspect variable; the value holder becomes the *model* of the widget.

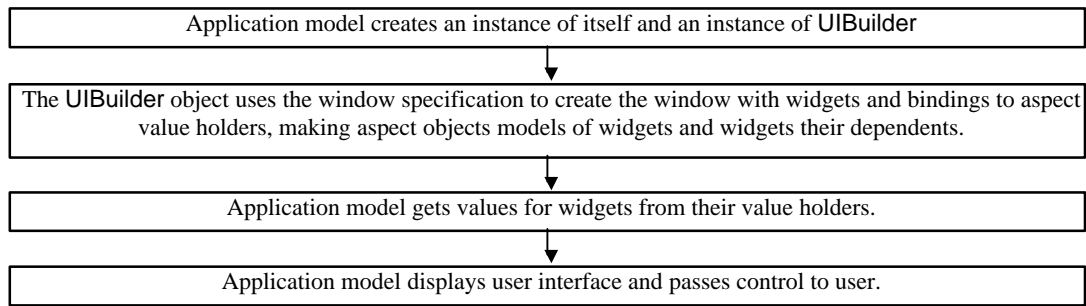


Figure 6.24. Essence of the application opening process.

Main lessons learned:

- Value-dependent widgets hold their value in a `ValueHolder`. This value holder is accessible via the widget's *Aspect* variable and method specified as the widget's property.
- Widgets and their value holders are bound by the model \leftrightarrow dependent relationship. The value holder is the model, the widget is the dependent.
- When the application changes the value of a value holder via the `value:` message, the value holder notifies all its dependents by sending them the `update: #value` message. The reaction of each dependent depends on its definition of `update:`. The definition is different for each widget.
- When the dependent is a widget with an *Aspect*, its reaction to `update: #value` is to send its *Aspect* message to its model, obtaining its new value, and redisplaying itself accordingly.
- To obtain the proper reaction of a widget to a change of its *Aspect* value, always change the value of the *Aspect* value holder by the `value:` message.
- Using assignment instead of the `value:` mechanism is the most common reason why widgets don't respond to value changes.
- Constructing the user interface from the window specification method is the responsibility of a `UIBuilder`. An application model creates its instance when it opens, and keeps it during the application's lifetime.

Exercises

1. Implement the material covered in this section.
2. Trace how the Text Editor in our application responds to clicking an application button and summarize your findings in a diagram. (Hint: Add `self halt` before the `value:` message in the button method.)
3. Class `ValueHolder` is rather low in the class hierarchy. Examine its superclasses and explain how `ValueHolder` obtains its behavior and what new features it adds.
4. Enact the following scenarios using a simple application such as `SimpleHelp` or `ApplicationSelector` as the application model.
 - a. Painting of the user interface and its installation. Creates application model with its *Aspect* and *Action* methods and variables, stores window specification. Actors are the developer of the user interface and an instance of `UIPainter`, the object that animates the user interface painting tools.
 - b. Opening the application. Actors are the user who sends the opening message, the application model class and its instance, a `UIBuilder`, and instances of widgets and their value holders.
 - c. Interaction with an open application. Actors are the user, the application model, the widgets, and their value holders.
5. A model may have several dependents. As an example, an application displaying mathematical functions might display the values in graphical and textual form (using a `Table` widget, for example). Both displays would be dependents of a value holder with a list of function values. Enact the basic operation of the dependency mechanism along the lines of part c of the previous exercise.

6. The model-dependent relationship has many applications beyond user interfaces. As an example, consider a collection of physical particles that works as follows: When a particle changes its energy by some amount Δ greater than Δ_{\min} , all its neighbors change their energy by $\Delta/2$, all their neighbors change their energy by $\Delta/4$, and so on. This reaction continues until it dies down when the transmitted energy falls below the lower limit. Describe this problem in terms of models and dependencies, and enact it.
7. We used the term *model* in three contexts: application model, domain model, and model in the model-dependent sense. Define the three meanings carefully to clarify their distinct meanings.

6.8 Opening of an application - hook methods

In this section, we will give a more detailed description of the events that take place when an application model class receives the open message (Figure 6.25). Understanding this sequence is essential for initialization and for other operations related to the user interface.

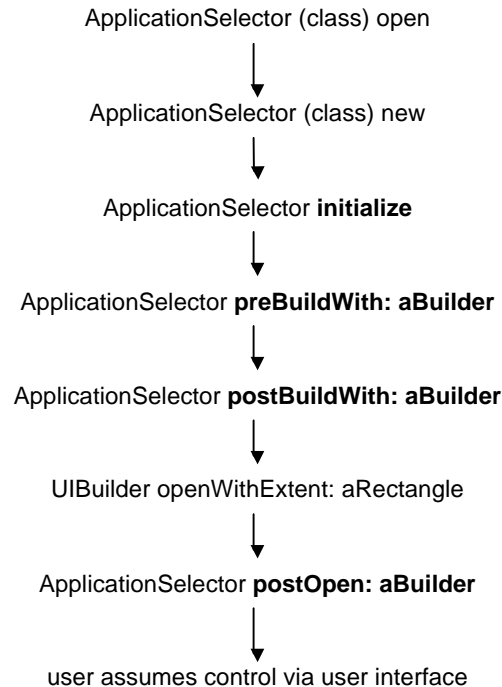


Figure 6.25. The main messages executed in response to `ApplicationSelector open`. Boldface messages are hooks that can be redefined in your application model.

To trace the sequence and to follow our description ‘live’, execute

`self halt.`

`ApplicationSelector open`

and note the following events:

1. Method `open` sends **new** which creates an instance of the application model class `ApplicationSelector`.
2. The application model `ApplicationSelector` sends itself the **initialize** message. If you defined this message in your application model, this definition is executed. If you did not, the inherited definition of `initialize` is executed. The default definition does not do anything and its only purpose is to intercept the

initialize message if it is not fielded by a subclass. This is an example of a *hook method* - a method that provides an opportunity for the developer to execute an operation at a critical place in the process.

3. ApplicationSelector now asks class UIBuilder to create a UIBuilder object. This object will eventually obtain the information about your interface from windowSpecs and 'build' it. First, however, the application model sends **preBuildWith: aBuilder** to itself; where aBuilder is the new builder object. If you defined preBuildWith: in your application model class, this method will now be executed, otherwise the 'do-nothing' default definition of preBuildWith: inherited from ApplicationModel will be executed. The method is another hook.
4. After executing preBuildWith: the builder builds the user interface in memory using windowSpec: It gathers information about the window's size and label and the widgets, and constructs their bindings and a UIPolicy object that will help build the window. The 'policy' object determines which style will be used to draw the window, allowing a choice between the Microsoft Windows look, the Macintosh look, and the Motif look. After this, the application model sends **postBuildWith: aBuilder** - another hook message. Again, the redefined version does not do anything.
5. The application model now draws the window on the screen, but before it passes control to the user, it sends **postOpenWith: aBuilder**. This is the last hook (its inherited behavior is to do nothing) and after it executes, the application opens for user input via widgets.

In your application, you can use any of the application opening hooks, all of them, or none of them. In some cases, the nature of what you want to do determines the exact hook that you must use, in other cases, the same effect can be achieved by using any one of several hooks. As an example, if you want to do something that depends on the existence of widgets, you must allow the builder to build the widgets first, which means that you cannot do this in the initialize or preBuildWith: methods.

In our application, we only needed a hook to assign a value holder with the initial text to the description variable. Although this can be done with any hook method, it is natural to do it during initialization in method initialize as we did:

initialize

```
"Create a value holder with the text in the initial Text Editor display."  
description := 'Please select an example application.' asValue
```

Pseudo-variable super

Although our definition of initialize works fine, it does not follow the recommended style for hooks. The point is that our application model class might be at a low point in the application model chain (such as class E in Figure 6.26) and some of its superclasses might contain their own definitions of initialize. Since our new class should normally execute this inherited behavior and augment it with its own behavior, we should define initialize to execute the inherited behavior first and the specialized behavior next as in

initialize

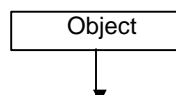
```
self initialize. "Execute inherited behavior."  
description := 'Please select an example application.' asValue
```

Unfortunately, this form would create an infinite loop, because it makes self send initialize to itself over and over, until we stop the loop with <Ctrl> <C>. Obviously, our goal was not to re-execute *this* definition of initialize but rather the definition higher up in the hierarchy tree. To locate this class, Smalltalk provides a special identifier called super and the correct way to define initialize using super is as follows:

initialize

```
super initialize. "Execute inherited behavior."  
description := 'Please select an example application.' asValue
```

The concept of super is very important and we will use Figure 6.26 to explain its exact meaning.



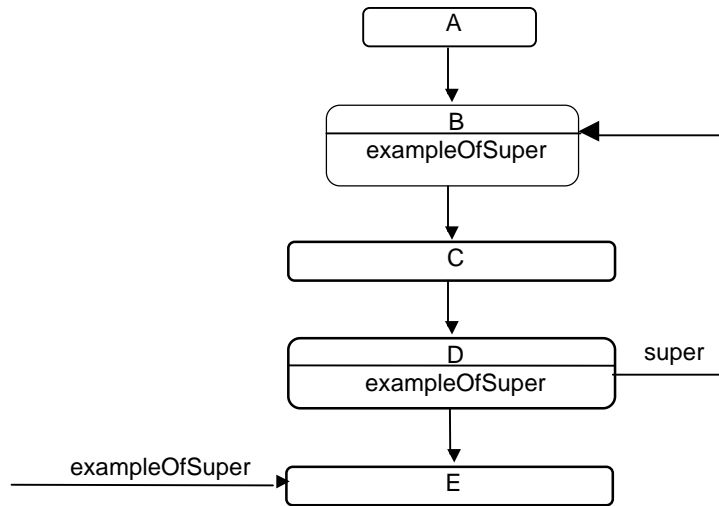


Figure 6.26. Example hierarchy illustrating the meaning of super.

Assume that class D contains method `exampleOfSuper` defined as follows:

exampleOfSuper

```
"some code here"  
super exampleOfSuper.  
"some more code"
```

When you send `exampleOfSuper` to an instance of class E, it will pass execution to the definition of `exampleOfSuper` in class D. In this definition, `super` will refer to the nearest superclass of D which contains its own definition of `exampleOfSuper`, in this case class B. A very common misconception is that `super` refers to the superclass of the receiver, in this case D.

Note the similarities and differences between `self` and `super`. In two ways, `super` is similar to `self`: Both are *like* a variable in that their referent may vary from one context to another, but they are *not really* variables because we cannot assign value to them. Because of this partial but incomplete similarity to variables, `self` and `super` are both referred to as *pseudo-variables* or *special variables*.

In another way, `super` and `self` are different: Pseudo-variable `self` represents an object - the receiver of the message. You can thus meaningfully write

```
^self
```

The purpose of `super`, on the other hand, is to specify a class containing the definition of a message. Without specifying this message, `super` does not make any sense and if you try to use `super` by itself, as in

```
^super
```

Smalltalk will refuse to compile the code (Figure 6.27).

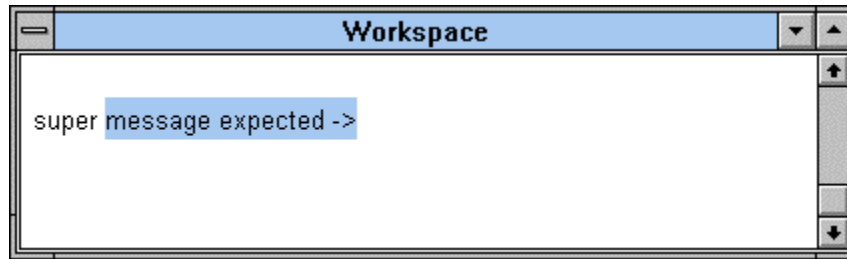


Figure 6.27. Pseudo - variable super must be followed by a message.

Main lessons learned:

- A hook is a method automatically executed during a process such as the opening or closing of a window. Its inherited definition usually does not do anything but since the developer of a subclass can redefine it, a hook makes it possible to insert code at strategic places.
- The opening sequence of ApplicationModel includes several 'do-nothing' hook messages.
- The proper way to redefine a hook message is to execute first its inherited version with super, and then execute the specialized behavior.
- self and super are pseudo-variables - the object that they refer to depends on the context but it cannot be changed by the programmer.
- Unlike self, super cannot stand on its own. It must always be used as the receiver of a message.

Exercises

1. Implement the material covered in this section.
2. Trace and record the complete sequence of message sends following the execution of open.

6.9 MVC – the Model-View-Controller triad

After opening the subject of widgets, it is time to explain the essence of GUI components in VisualWorks Smalltalk. The principle of user interface components in Smalltalk is the *model-view-controller (MVC) paradigm*. According to this principle, every UI component that represents a value of a domain object uses this value as its *model*. As an example, a text editor's model is the displayed text, a slider's model is a floating-point number, and the model of an on/off check box model is a Boolean. The model holds the data but knows nothing about its display and user interaction, leaving these responsibilities to the view-controller pair.

The graphical representation of the model is the responsibility of a *view* object. As an example, a text may be displayed by the view of a text editor, a floating point number may be displayed by the view of a slider, and an image may be displayed by a custom view. The view object knows how to display itself from data supplied by its model but does not hold the original of the data and is incapable of user interaction.

Finally, the object that manages user activity within the boundaries of a view is the view's *controller*. A controller is notified when an input event such as a mouse click occurs, and defines methods that process these events. As an example, when a user clicks the <select> button to select text inside a text editor, the editor's controller is notified and converts these events into a highlighted text selection in cooperation with its model and view. A controller knows nothing about the data and its display but it is in full control of the user interface.

Each of the three components of the MVC triad is thus in charge of one aspect of UI operation and the three cooperate to accomplish all the tasks that we expect of the user interface - display, damage repair (such as when a window is obscured by another window or collapsed and expanded), automatic response to domain changes, and response to user actions. Besides the conceptual elegance of this arrangement, this separation of responsibility enhances flexibility and reusability because any of the three components can be

combined with others in new combinations, thereby eliminating the need to construct a variety of custom user interfaces with similar behaviors. Some of the possibilities are as follows:

- The same model data may be displayed by a different view in different modes of operation of the application. As an example, an object representing economic data may be represented as a pie chart or as a graph and the switch from one to another may be controlled by clicking a button.
- The same model may be simultaneously displayed by several different views. As an example, a mathematical function may be displayed as a collection of x and y values in a table, and as a diagram in the same window.
- The same view can be used with many different models. As an example, a pie chart object may represent economic data in one application and demographic data in another without any change in its definition. The switch requires only the assignment of a different model.
- The same view can use different controllers. As an example, the same Text Editor may use a read-write controller when the application is used by fully authorized users, and a read-only controller when the program is used by less authorized users.
- The same controller may be used with different views. As an example, a controller sensitive to mouse clicks and equipped with an <operate> menu could be used in an input field and a text editor.

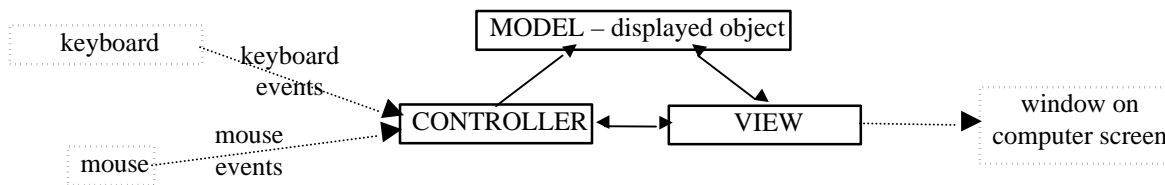
Smalltalk's implementation of the MVC concept is based on the communication patterns in Figure 6.8 and made possible by instance variables of the three objects. As the diagram shows, each object in the triad can communicate directly with each other object except that the model does not have direct access to the controller view. The typical communications are as follows:

The *controller* generally needs to notify the *model* about user interaction. As an example, when the user clicks a square in a clickable chess board view in a chess program (Section 12.7), the controller must tell the model which square has been selected.

The *view* needs access to its *model* when it needs to redisplay itself. As an example, when a chess player clicks a piece, the controller notifies the model, the chess board model asks the view to display it, and the view asks the model for information about the squares that must be redisplayed. This typical scenario thus requires controller \rightarrow model and model \leftrightarrow view communication.

As an example of communication between a *controller* and its *view*, consider an application such as the canvas painter in which the user can drag objects within a window. To do this, the controller can communicate with the view directly.

Communication from the *view* to the *controller* is usually restricted to specifying an appropriate controller class when the first input event occurs. This message then creates the controller.



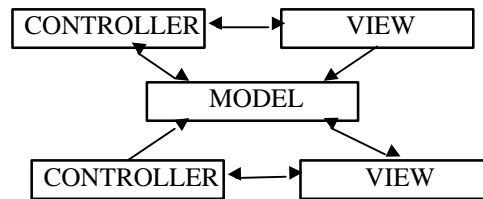


Figure 6.8. Top: Standard lines of communication among the three components of the MVC triad and the meaning of MVC components. Bottom: A single model may have two or more different view-controller pairs.

Although every widget that displays data has its special model, view, and controller, it is not surprising that all models, all views, and all controllers have something in common and that their implementation is based on abstract class called Model, View, and Controller. We will now outline the principle of these class and return to views and controllers in much more detail in Chapter 12 which is fully dedicated to this subject.

Class Model is responsible for the dependency mechanism. Its definition is quite simple and consists of variable dependents which can hold any number of objects dependent on the model, and methods implementing dependency. (In fact, these methods are inherited from Object.) Class Model has numerous subclasses including ApplicationModel (and with it all application models defined by the system or the user) and ValueHolder (a subclass of ValueModel which defines value, value: and setValue:).

Class Controller is the father of all controllers. It introduces variables view and model that refer to its companions in the MVC triad, and numerous methods. Perhaps the most interesting of these are methods that provide hooks to input events and are sent when a user clicks a button, moves the mouse, presses a key, and so on. These methods are hooks that are reused by all subclasses whether system-defined or user-defined.

Class View is the mother of all view. It knows about its model and controller and provides or inherits mechanisms for creating nested views. It also provides automatic damage repair which is necessary when the view is damaged (for example obstructed and then uncovered). Any class subclassed to View inherits this essential and valuable behavior.

We will have much more to say about the MVC triad and about its components in Chapter 12 which is completely dedicated to views and controllers.

12.7 IDs make widgets accessible at run time - a Tic-Tac-Toe game

In this section, we will explain the purpose of widget IDs and illustrate it on a program implementing a Tic-Tac-Toe game with the user interface in Figure 6.28.

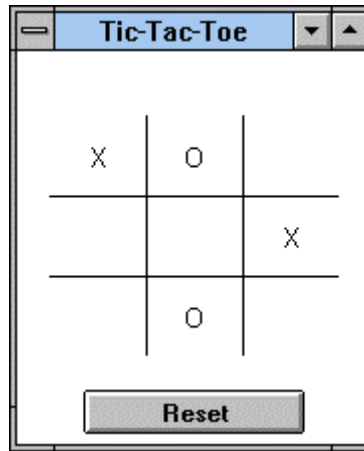


Figure 6.28. Desired user interface.

The game is played by two players denoted X and O; the first player is always X. When the window first opens, the squares on the game board are blank and the players start taking turns clicking the squares. When a player clicks an empty square, the square displays the player's symbol (X or O). If this results in three vertical, three horizontal, or three diagonal squares marked by the same symbol, the player wins; otherwise the game continues. If the player clicks an already marked square, the program displays a warning asking the user to click an empty square. The *Reset* button resets the board to blank squares. The usage scenarios are as follows:

Scenario 1: Player clicks empty square.

Conversation:

1. *User* clicks square.
2. *System* displays player's symbol.
3. *System* checks whether the game is over. If it is, it displays an appropriate notification and resets the board; if it is not, it changes current player from X to O or vice versa.

Scenario 2: Player clicks square that already contains a symbol.

Conversation:

1. *User* clicks square.
2. *System* displays a warning asking the player to click an empty square.

Scenario 3: Player clicks *Reset*.

Conversation:

1. *User* clicks *Reset*.
2. *System* erases all squares.
3. *System* resets player symbol to X.

Design

The problem is simple and we can design the solution without going into much detail. The domain objects consist of players and the grid. The only thing the program needs to know about the players is who is the current player (X or O) and this requires only a String object. The only other required class is the application model.

All in all, we don't need any domain classes and the only necessary information is instance variable *player* holding the string identifying the current player. Our program will thus consist of a single class, an application model class called *TicTacToe*, a subclass of *ApplicationModel*. Its behaviors will be as follows:

- initialization - initialize player to 'X'
- actions - respond to activation of board squares and the *Reset* button
- private - check for end of game, toggle players after a move

Implementation

We will implement the squares as action buttons with blanks, Xs or Os as labels. We will need to change button labels at run time as the players click them, and to do this, we need run time access to them. This access can be gained via the builder which holds a dictionary of all *named widgets* (widgets assigned IDs with the Properties Tool) which associates widget IDs and the corresponding value holders. To make a widget accessible fill in its ID property; to access it, ask the builder to locate this component via its ID. As an example, if we assigned the upper left action button the ID `button1`, we can access it at run time by

`builder componentAt: #button1`

To assign IDs, use the *ID* field in the Properties Tool as in Figure 6.29. Any widget may have an ID.

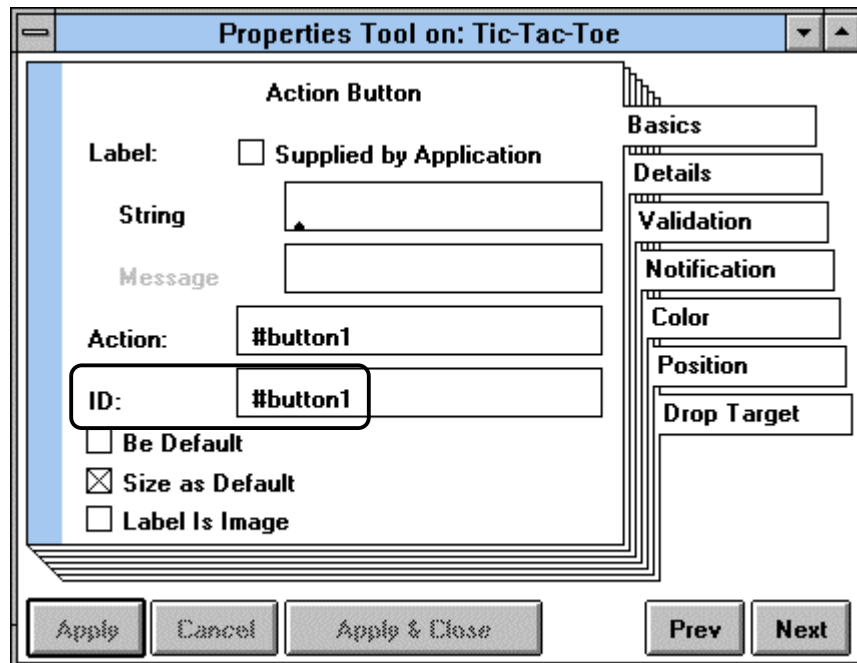


Figure 6.29. Assigning an ID to a widget so that it can be accessed at run time.

Now that we know how to access a widget, how can we access its label? First of all, the component accessed by the builder is not really the widget but a *wrapper* object containing the widget. To get the widget, send the widget message to the wrapper. After this, ask the widget for its label, and then access the label's text. Altogether, to get a widget's label, execute

```
(aBuilder componentAt: #button1) widget label text    "Get text of label of button1."
```

To change the text of the label, use message `labelString:`.

We are now ready to start implementing the program. First, paint the canvas and name the button IDs `#button1`, `#button2`, and so on. Since the window should open with empty squares, the *String* assigned to button labels in the Property Tool will be empty (Figure 6.29). When the user clicks `button1`, its *Action* method will check whether the label is an empty string and respond as follows:

button1

```
"Button was clicked. Check if it is already labeled and respond appropriately."  
self builder componentAt: #button1 widget label text isEmpty  
  ifTrue: ["Get component, change its label, exit if game is over, switch players."  
    self builder componentAt: #button1 widget label labelString: player.  
    self endOfGame ifTrue: [^Dialog warn: 'Game over.'].  
    self newPlayer]  
  ifFalse: [Dialog warn: 'This field is already occupied']
```

This works but repeating expression

```
self builder componentAt: #button1 widget
```

is ugly. Moreover, it will have to be accessed again in the *reset Action* method and we thus decide to evaluate the expression once for all during initialization and save (cache) the result in an instance variable as

```
button1 := self builder componentAt: #button1 widget
```

Assuming that this has been done, the definition becomes

button1

```
"Button was clicked. Check if it is already labeled and respond appropriately."  
  button1 text isEmpty  
    ifTrue: [button1 text: player.  
      self endOfGame ifTrue: [^self].  
      self newPlayer]  
    ifFalse: [Dialog warn: 'This field is already occupied']
```

This definition must be repeated with minor variations for each button and this is not very nice. We will give a better solution at the end of this section.

The next question is where to put the assignment to *button1* and the variables corresponding to the other buttons. Clearly, it must be before the application opens, in one of the hook methods. It cannot be done in *initialize* because the builder does not yet exist, it cannot be done in *preBuildWith:* because the builder has not yet processed widget properties and does not have the ID dictionary. We will thus do it in *postBuildWith:* as follows:

postBuildWith: aBuilder

```
"Cache button labels for easy access in Action methods."  
  button1 := (aBuilder componentAt: #button1) widget label.  
  button2 := (aBuilder componentAt: #button2) widget label.  
  button3 := (aBuilder componentAt: #button3) widget label.  
  button4 := (aBuilder componentAt: #button4) widget label.  
  button5 := (aBuilder componentAt: #button5) widget label.  
  button6 := (aBuilder componentAt: #button6) widget label.  
  button7 := (aBuilder componentAt: #button7) widget label.  
  button8 := (aBuilder componentAt: #button8) widget label.  
  button9 := (aBuilder componentAt: #button9) widget label
```

As the next step, we will implement *endOfGame* which checks for end of game. Our implementation will be very straightforward - we will simply check whether any of the rows, columns, or diagonals are filled with copies of the current player's symbol, and return the *or* of this combination. The definition is as follows:

endOfGame

```
"Check all rows and columns and diagonals for end of game. Return true or false."  
| end |  
end := ((self isPlayer: button1) & (self isPlayer: button2) & (self isPlayer: button3)) |  
  ((self isPlayer: button4) & (self isPlayer: button5) & (self isPlayer: button6)) |  
  ((self isPlayer: button7) & (self isPlayer: button8) & (self isPlayer: button9)) |
```



```
((self isPlayer: button1) & (self isPlayer: button4) & (self isPlayer: button7)) |  
((self isPlayer: button2) & (self isPlayer: button5) & (self isPlayer: button8)) |  
((self isPlayer: button3) & (self isPlayer: button6) & (self isPlayer: button9)) |  
((self isPlayer: button1) & (self isPlayer: button5) & (self isPlayer: button9)) |  
((self isPlayer: button3) & (self isPlayer: button5) & (self isPlayer: button7)).  
end ifTrue: [self reset. Dialog warn: 'Player ', player, ' wins. Game over.'].  
^end
```

where we used fully evaluating logic because it is more readable and because evaluation speed is irrelevant in this case. Method `isPlayer: aButton` checks whether `aButton`'s symbol is equal to the player's symbol as follows:

isPlayer: button

```
"Is the symbol displayed in button the same as the player's symbol?"  
^button label text = player asText
```

where we had to convert the player string because a label's text is a `Text` object which is somewhat different from a string and cannot be directly compared with it. We will learn about `Text` later.

Finally, method `newPlayer` toggles the player:

newPlayer

```
"Update player."  
player = 'X'  
    ifTrue: [player := 'O']  
    ifFalse: [player := 'X']
```

Finally, the `reset` method. It resets all button labels and initializes player to 'X':

reset

```
"Reset button and reinitialize the player."  
button1 labelString: ".  
button2 labelString: ".  
button3 labelString: ".  
button4 labelString: ".  
button5 labelString: ".  
button6 labelString: ".  
button7 labelString: ".  
button8 labelString: ".  
button9 labelString: ".  
player := 'X'
```

Improving button methods

When we wrote the `button1` method, we noted that it is repeated with very minor variations for all buttons and this is ugly. Besides, if we decide to make any changes, we will have to repeat them in all nine methods. We will thus change all button methods to the following style

button1

```
self doButton: (builder componentAt: #button1)
```

and put all the shared code in

doButton: aButton

```
"Game button aButton has been clicked. Check if the button is already in use and if not, change its label,  
check for end of game, and switch players if not done."
```

```
aButton widget label text isEmpty  
    ifTrue:  
        [aButton widget labelString: player.  
        self endOfGame ifTrue: [^self].
```

```
self newPlayer]
ifFalse: [Dialog warn: 'This field is already occupied']
```

Note that as we are changing the state of the object assigned to variables `button1`, `button2`, etc., we don't have to make any changes to the variables themselves – they keep pointing to 'the same' object whose properties have changed via `doButton`:

You might argue that method `reset` has a similar problem as our button methods – it seems unnecessarily repetitive and it should be possible to simplify it. We will see in Chapter 7 that this can indeed be done.

Main lessons learned:

- If you want to be able to refer to a widget at run time, assign it an ID property and use the builder to access it via `componentAt: message`.
- A widget in a running application is enclosed in a wrapper. To access the widget, send `widget` to the component.

Exercises

1. Complete and test the implementation of class `TicTacToe`.
2. Our implementation of `endOfGame` ignores the possibility that the game ends in a draw. Correct this shortcoming.
3. Improve `TicTacToe` by adding a button to display the score of the two players in a notifier window.
4. Formulate a strategy for the computer to be one of the players. The strategy may be very simple such as choosing a square randomly, or more sophisticated.
5. Open `TicTacToe` in step-by-step mode and inspect the builder. List the contents of its `namedComponents` dictionary.
6. Our application can be implemented more neatly by defining domain model class `GameBoard` implementing all the functionality except the user interface, and implementing the GUI by a simplified version of `TicTacToe` with no domain behavior. Reimplement the program along these lines.

Conclusion

This chapter introduced development of applications with graphical user interfaces. VisualWorks' application architecture has three components: the graphical user interface (GUI), the application model, and the domain model. The role of the GUI is to display results and provide means of interaction between the user and the computer. The domain model is a collection of classes representing objects in the problem world. The application model provides the link between the GUI and the domain model, converting user actions into messages to domain objects, and changes in domain objects to messages to the user interface. Most applications contain all three parts but very simple applications combine the domain model and the application model in one class. Occasionally, an application does not have a GUI and does not require an application model either. Such applications are called 'headless'.

To minimize the effort required to create an application, VisualWorks provides a library of GUI components, class `ApplicationModel` containing shared properties of application models, and tools for interactive creation of the user interface with minimal programming. To take full advantage of the built-in functionality, application models must be subclasses of `ApplicationModel`.

VisualWorks UI development environment is based on three tools: a canvas (the future window), a Palette of widget buttons, and a Canvas Tool. To create a user interface, the programmer selects widgets on the palette, paints them on the canvas, and defines their properties such as labels, colors, and actions using the Properties Tool.

During the process of creating a user interface, the programmer installs the GUI on the application model which stores a description of the GUI in a class method of the application model. The *Define* command of the Canvas Tool defines aspect variables and stubs of *Action* and *Aspect* methods.

The principle of updating value holding widgets is dependence between the value objects and the widgets themselves. This dependence is achieved by storing the values in instances of *ValueHolder* whose instances provide a uniform interface to their value components and implement dependency. When the value of a *ValueHolder* is changed with the *value:* message, the *ValueHolder* notifies its dependents and they respond by executing their predefined behavior. A Text Editor widget, for example, responds by using its *Aspect* message to request text from the application model and by redisplaying itself with the new text.

When an application model class receives the *open* message, it executes a series of messages including the building of the user interface by a *UIBuilder* object and the opening of the window from this 'built' before it transfers control to the user interface. The opening sequence includes several hooks - *do-nothing* messages defined in *ApplicationModel* to allow the programmer to insert any appropriate actions into the application opening process.

Any widget can be assigned an ID and this ID can be used to access the widget at run time for effects such as changes of labels and other effects.

The basis of VisualWorks graphical user interfaces is the MVC paradigm in which an object holding data (the model) is displayed by a view and the user interface is implemented by a controller. Class *View* defines several mechanism for keeping views up to date, and new view classes created for new GUI components should thus be subclassed to *View*.

The controller part of MVC is responsible for mouse- and keyboard-based user interaction within the view's area. The preferred method for propagating model changes to the view is via dependency where each significant change of the model notifies all model's dependents. In the MVC triad, the dependent is the view.

As a side product of developing our example applications, we introduced the very important concept of the pseudo-variable *super*. The purpose of *super* is to provide access to a higher level definition of a method, usually to avoid recursion in a new method with the same name. Pseudo-variable *super* does not represent an object and cannot be used by itself. It must always be used as a receiver of a message.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

ApplicationModel, *SimpleHelp*, **UIBuilder**, **ValueHolder**.

Terms introduced in this chapter

active widget - a widget capable of interaction with the user

application model - object linking user interface and domain model; subclassed from *ApplicationModel*

Action button - UI button that executes a predefined action when clicked

Action method - action button property; method defined in application model and invoked when user clicks the button

Aspect method - widget property; application model method used to access the value of the widget's value holder

Aspect variable - application model variable bound to a *ValueHolder* holding the model of a widget

builder - instance of *UIBuilder*, part of VisualWorks framework which constructs the user interface before a window opens, and provides access to UI components during execution

canvas - blueprint of a future window on which widgets are painted during user interface design

Canvas Tool - window providing access to commands during the construction of the user interface

controller - the object responsible for dealing with user input in an MVC triad

Define Tool - tool for automatic definition of *Action* and *Aspect* methods and aspect variables

dependency - model ↔ dependent relationship in which the model automatically notifies its dependents of its changes

domain model - collection of classes representing objects in the problem domain

graphical user interface (GUI) - interface between an application and its human user implemented with windows and widgets

GUI - graphical user interface

hook - method built into a process such as application opening or closing to allow developers to insert application-specific actions into the process

input event - operation such as pressing a key on the keyboard, moving the mouse, or clicking a mouse button

Install - UI Painter command; defines the application model class if it does not exist and stores the GUI description in a user-specified class method

lazy initialization - leaving initialization of an instance variable until the time when its value is requested

MVC paradigm - model - view - controller - the three parts of VisualWorks user interfaces responsible for the data, its display, and user interaction respectively

model - the controlling part of the model \leftrightarrow dependent relationship

model \leftrightarrow dependent relationship - see *dependency*

passive widget - a widget that does not allow user interaction

Properties Tool - tool allowing specification of window and widget properties such as labels, names of *Action* and *Aspect* methods, colors, and so on

pseudo-variable - identifier whose meaning depends on the context but whose value cannot be changed by assignment; *self* and *super* are pseudo-variables

super - pseudo-variable providing access to a higher level definition of a method

Text Editor - active GUI widget displaying text and allowing user input

UI builder - user interface builder - see *builder*

UI Painter - interactive tool for GUI development with minimal programming

value holder - instance of *ValueHolder*, an object holding a value and a list of dependents; used as model in the model \leftrightarrow dependent relationships

view - the object responsible for displaying the model in the MVC triad

widget - a passive or active GUI component such as an *Action Button*, a *Label*, or a *List*

Chapter 7 - Introduction to Collections

Overview

Catalogs, inventories, dictionaries, lists of popup menu commands, items in a shopping cart - all these are examples of the fact that objects almost always occur in groups. Because collections of objects are so essential and varied, the Smalltalk library contains many collection classes and numerous useful methods for manipulating them.

In this chapter, we will survey the general properties of the collection class hierarchy and explore arrays, the basic kind of 'ordered' collections whose elements are stored in a fixed order and accessible by an index. We will also examine some extensions of the concept of an array, and the table widget as its application. The remaining collections will be covered in the following chapters.

The last section of this chapter introduces a new GUI widget – the Table. We encourage you to complement this material by reading Appendix 1 which contains a description of additional widgets, explains in more detail how widgets work, and includes several interesting examples.

7.1. Introduction

Although the world consists of individual objects, objects most often occur in groups. Some examples are bicycles standing in a school yard, a sequence of pages in a book, cars following one another on a highway, customers lined up in a bank, a bag of items bought in a store, a team of hockey players, and the seats in an auditorium.

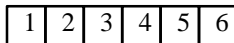
Given this reality, it is not surprising that most programming problems objects also deal with collections of objects. As an example, the text on a page is a string of letters, digits, and punctuation symbols. List widgets, pop up menus, drop down menus, tables, and other user interface components also display collections of items. Other examples include an airport display of flight departures, an inventory program managing a collection of inventory items, and a course-marking program managing a collection of student records, themselves collections of marks and other information. Finally, a computerized encyclopedia provides access to a collection of keywords and their definitions. Several of these examples show that the elements of some collections are themselves collections and it thus makes perfect sense to treat collections themselves as objects.

A closer look shows that collections come in a variety of forms (Figure 7.1):

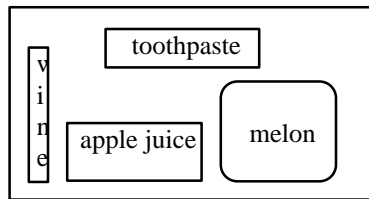
- Each player on a hockey team has a unique number and the number of players on the team is fixed. We could line the players up in the order of their numbers and we could refer to each by his or her number. If the numbers were consecutive integers, we could put the players into consecutive numbered slots and access them by these numbers. A line up of swimmers on starting blocks is an even better illustration of this kind of collection. In the programming context, this kind of collection is called an *array* and its characteristics are that it has a fixed size and its elements can be accessed by an integer 'index'.
- Cars on a parking lot can also be associated with numbers. The area could be divided into rows and columns, consecutive rows and columns assigned numbers, and each car identified by its row and column numbers. This collection of slots and their contents (cars) is again an array because it has a fixed size and can be accessed by a 'key', but it has two dimensions - rows and columns. A parking lot is an example of a *two-dimensional array*.
- Cars on a highway are a different kind of collection because their number constantly changes as cars arrive on the highway and leave it through exits. In Smalltalk, this kind of collection is called an *ordered collection* and its characteristics are that the number of elements in an ordered collection can change and that the contents are ordered: Assuming a one-lane highway, the cars follow one another. In principle, we can again access the cars by an index but since new cars are added all the time at one end, and leave at the other end, this is not the most natural way to refer to the elements of an ordered collection. The first, the last, and the next or the previous element are usually more important concepts.

- An encyclopedia is yet another kind of collection because its elements themselves have a structure - each consists of a keyword and a value (such as the definition of a word). This kind of collection occurs so often that Smalltalk treats it as a special kind of collection called a *dictionary*. Elements of dictionaries are usually accessed by their key as in 'what does the word *xenophobia* mean?'
- The structure of a collection of items in a shopping bag is very different from the previous examples because its contents are *not* ordered. In Smalltalk, this kind of collection is rather predictably called a *bag*. An interesting property of the shopping bag example is that each item in the bag often is of a different kind such as tooth paste, apple juice, and melon. In fact, most Smalltalk collections allow their elements to be arbitrary objects.
How are the elements in a bag accessed? Some people just throw them in the bag to settle wherever there is place for them, and when you pull them out without looking, they come out in an unpredictable order.

starting blocks in a swimming pool



a shopping bag



parking lot arranged by slots
(row/column)

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3
4,1	4,2	4,3

a Czech – English dictionary

secese	– secession
sedm	– seven
sehnat	– to come by
sejit se	– to meet
sejmout	– to take off
sekera	– axe
sele	– sucking pig
sem	– hither
semafor	– semaphore
sem tam	– occasionally

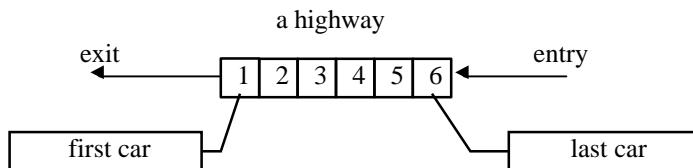


Figure 7.1. Collections may have different structures.

Our programming examples illustrate that collections come in various forms:

- A text stored in a document is a string of letters, digits, and punctuation symbols, in other words an ordered collection of characters.
- List widgets display one string on each line and since strings are ordered collections, a list widget is an ordered collection of ordered collections!
- The contents of a display of flight departures at an airport is an ordered collection of flight descriptions, and flight descriptions themselves are complicated objects with many components such as flight number, flight destination, departure time, and so on. In fact, since flight descriptions are arranged by their departure times, it is better to think of them as *sorted* rather than ordered collections: A *sorted collection* is just like an ordered collection, but its elements are sorted on the basis of some calculation. In our case, the calculation is comparison of departure times.

It is interesting to note that the nature of individual items in the display of departing flights can be interpreted in two ways: One is to view the individual items of the flight display (flight number, flight destination, departure time, and so on) as items of a 'flight object' collection. On the other hand, components of flight items are accessed by name as in 'what is your flight number?' or 'what is your departure time?' and it thus seems more natural to treat each flight item as an instance of a 'flight information' object. The second interpretation seems more convenient because if we treat flight information

as a collection, its elements are nameless and we can only ask questions such as ‘what is the value of the third element of the flight information collection’ - when we mean ‘what is the departure time?’. This shows that the decision whether to treat a complex object as a collection or an instance of a special kind of object may not be obvious until we start thinking about how we want to use this object.

We have just seen that one distinction between a collection and a composite object is that collection elements are accessed anonymously whereas elements of composite objects are accessed by a name. Another distinction between a collection object and a composite object is that a collection can have any number of components but an object consisting of a fixed number of named components can only have the predetermined fixed number of components. As an example, the number of cars on a highway depends on the current situation. Even the number of slots in a parking lot is arbitrary because a parking lot can be designed any size or shape we want. A flight object treated as an instance of a composite object `Flight`, on the other hand, may only have a `flightNumber` component, a `FlightDestination` component, and a `DepartureTime` component if this is how the object `Flight` is defined.

The fact that collection elements are not named has the advantage that they can be accessed more easily. As an example, it is easy to describe how to calculate the total price of all elements in the shopping bag: ‘Take the elements in the bag one after another and add their prices.’ This procedure will work whether the bag contains one element or 100 elements, and whether the items are oranges, dishwasher liquid bottles, soap, or anything else. Similarly, counting all trucks in a satellite snapshot of a highway can be described: ‘Start with the first element and count one element after another until you reach the last element.’

In the case of a composite object with named components, enumeration (execution of the same operation on each component) is more complicated and requires explicit naming. Assume, as an example, that we refer to the 11 players on a soccer team as `goalie`, `leftDefence`, `rightDefence`, and so on. If we wanted to calculate the average weight of all players on the team, we would have to calculate

`averageWeight := (goalie weight + leftDefence weight + etc.) / 11`

naming each player explicitly. If we put player information into an array, the solution is simply

`averageWeight := (sum of weights of elements 1 to 11) / 11`

The obvious need for collections, their open nature, and the generality of enumeration over collections are their essential advantages that make them members of the family of essential Smalltalk classes.

Main lessons learned:

- Objects usually occur in groups. In Smalltalk, such groups are called collections.
- Careful examination shows that different situations require different kinds of collections such as ordered or unordered and fixed or variable sized.
- Performing the same operation on each element of a collection is called enumeration. Enumeration is one of the most important operations on collections.
- Smalltalk objects can be divided in two categories - those that contain zero or more named objects, and those that contain a variable number of nameless objects. Collection items are nameless and this has two major advantages:
 - A collection may contain any number of elements whereas an object with named components contains only the named components.
 - It is much easier to enumerate over unnamed elements of a collection than over named instance variables of a multi-component object.
- Collections belong to the most important programming concepts.

Exercises

1. What kind of collection or named object is most natural for the following situations:
 - a. Customers waiting in a bank queue.
 - b. Filing cabinet drawer with alphabetically arranged current student files.
 - c. Book information including author, title, and catalog number.
 - d. Filing cabinet drawer with files of students who graduated last year.
 - e. Description of all trees in the city of Halifax.
 - f. A list of all rooms on one floor in a student residence.
 - g. Student information including first name, last name, home address, home phone number, local address, local phone number, year of first registration, courses taken each past year, courses being taken this year, major area of study, previous degrees, participation in coop program.
 - h. A list of rooms in one residence building.
 - i. Bank account containing customer name, address, telephone number, and a list of accounts.
 - j. A chess board.
 - k. A list of 'print jobs' waiting to be printed on a shared printer.
2. What is the nature of the components of the collections and composite objects in Exercise 1?

7.2 Essential collections

Since collections are so useful and their applications so varied, the Smalltalk library contains more than 70 predefined collection classes. Some of them are abstract because different kinds of collections share many properties, but most are concrete. In the library, collections are divided into nine categories according to their abstractness, distinguishing characteristics, and uses. Many collection classes support the Smalltalk environment itself, but others are intended for general use. In the rest of this section, we will identify the essential collections (Figure 7.2) and give a brief overview of each.

Object

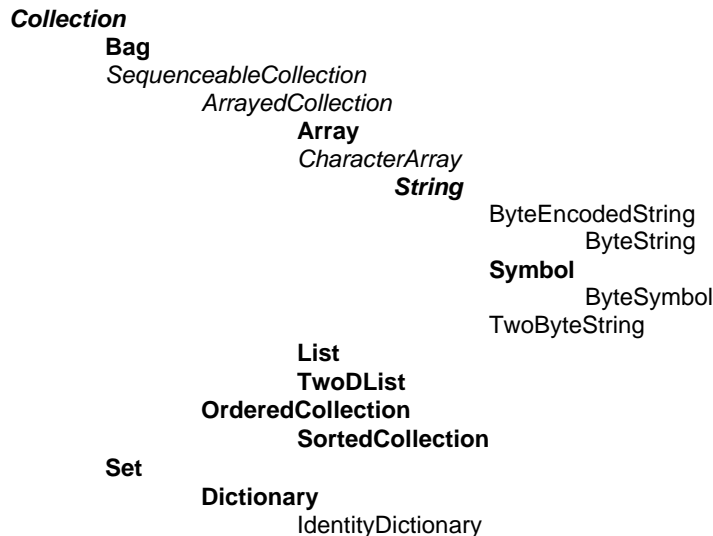


Figure 7.2. The most commonly used classes of the Collection hierarchy. Classes that will be covered in more detail are **boldfaced**, abstract classes are *italicized*.

Collections can be classified according to several criteria. One of them is whether their elements can be accessed by a key or not. A very important part of this group can access its elements by an integer number corresponding to the position of the element in the collection. These collections are called *sequenceable* and are implemented on a special branch of the class tree (Figure 7.2). Many of the collections whose elements are accessed by a non-integer key share the abstract class *KeyedCollection* as

their superclass and their branch is separate from sequenceable collections. These classes, however, are used mainly in Smalltalk implementation rather than general applications. Ironically, the most frequently used members of the family of collections accessed by a non-integer key (various kinds of dictionaries) is completely separate, and resides on the branch of classes that are not accessible by a key at all. This is because of the way they are implemented and because of the limitations of single inheritance.

Some collections (such as bags and sets) cannot be accessed by a key and Smalltalk calls them *unordered*. Although their elements are internally stored in consecutive memory locations, this internal arrangement is hidden from the user.

Another distinguishing feature of collections is that some collections have a *fixed size* which is established when the collection is created, and other collections can grow. Most Smalltalk collections automatically grow when they need more room to accommodate a new element.

Another distinction is that some collections allow *duplicates* whereas other collections throw duplicates away and keep just one copy of each element.

Finally, some sequenceable collections are *sorted* according to a predefined sorting function whereas other sequenceable collections are arranged according to the order in which the elements are added. In some sequenceable collections, the absolute position of an element may change when a new element is added or when an existing element is deleted, in other collections the position of an element never changes.

After this general overview, we will now review the main concrete collections before examining each of them in more detail later.

Arrays have fixed size determined by their creation message which has the form 'create an array of size n'. Their elements are ordered in sequence and accessed by an integer index. The index of the first element is always 1 (some other languages start at 0) and the following elements have consecutive indexes; as an example, elements of an array with five elements are numbered 1, 2, 3, 4, and 5. Once an object is assigned to a position, its position never changes unless this is explicitly requested by the program. Elements of Smalltalk arrays and most other collections may be any objects, and even a heterogeneous collection of objects of different kinds is acceptable. Because arrays have a fixed size, they are used when we know exactly how many elements the collection will have. Their advantage is that their accessing is very efficient and that they are often easier to create and initialize than other types of collections.

Ordered collections are also sequenceable which means that their elements are also ordered in a sequence and indexed. Unlike arrays, however, ordered collections have a built-in mechanism for growing. Also, ordered collections often add elements at the beginning and this changes the index of elements already in the collection. Programs using ordered collections in this way thus cannot use the index of an element to access it. The price for the flexibility of ordered collections is that some operations on them are not as efficient as operations on arrays. Automatic growing, in particular, which occurs when the currently allocated size becomes too small, is time-consuming. Ordered collections are used when we need to collect objects whose total number is variable or initially unknown and when the order in which they are received matters as when we simulate a queue of customers in a bank.

Sorted collections are a special kind of ordered collections that insert new elements into a position determined by some sorting function, typically comparison by magnitude. For greater flexibility, the sorting function can be specified or changed by the program. The ability to sort is very valuable but its price is that inserting a new element may require considerable processing.

Lists are a relatively recent addition to VisualWorks library and according to the manual, they are intended to supersede ordered and sorted collections. In essence, lists are ordered collections that understand sorting messages and implement the concept of dependency which makes them useful as item holders for widgets such as list views. This is, in fact, the main use of lists.

Two-dimensional lists are a variation on lists and arrays. Like arrays, they have fixed size and their elements are accessed by index or rather a pair of indices, one for row and one for column. They are used to support table widgets because they implement dependency.

Strings are indexed sequences of characters with no information about how they should be displayed. We have already seen that strings are among the most frequently used Smalltalk objects.

Text is an object combining the concept of a string with information about its rendering - the font, style, size, and color in which the string will be displayed on the screen or printed.

Whereas the collections listed above are all sequenceable, the following collections are unordered. We have already explained that this means that although their elements are internally stored in a sequence, the order is determined by the implementation rather than by the accessing message and is hidden from the user.

Sets are unordered collections that don't allow duplication. In other words, if you add an element that is already in the set, the set does not change. This property makes it possible to weed out duplicates from an existing collection simply by converting it to a set.

Bags are unordered just like sets but they keep a tally of the number of occurrences of their elements. If the same element is inserted into a bag five times, for example, the bag knows that its number of occurrences. When an element is removed, its count is decremented by 1 and when the count reaches zero, the element is removed. In contrast, if you remove an element from a set, it is gone, no matter how many times it has been added before.

Dictionaries are collections in which values are accessed by a key. The key may be any object and if we used integers for keys, we would essentially obtain ordered collections. Dictionaries are therefore conceptually generalized ordered collections. In terms of implementation, however, dictionaries are implemented as a subclass of Set. In practice, elements of dictionaries are usually Association objects, key - value pairs similar to entries in a dictionary or an encyclopedia. Being sets, dictionaries eliminate duplication using keys to perform the comparison. This means that adding an association whose key is already in the dictionary replaces the original association. Because the structure of dictionaries is different from the structure of other collections (by relying on key and value access), dictionaries provide a number of specialized messages.

IdentitySet and IdentityDictionary are important subclasses of Set and Dictionary that use == instead of = to eliminate duplication.

Main lessons learned:

- VisualWorks contains many collection classes, some for system needs but many for general use.
- The main differences between collections are whether their size is fixed or not, whether they allow duplication of elements, whether they can be accessed by a key, and whether their elements are ordered and how this ordering is achieved.
- Collections most useful for general use are arrays, ordered and sorted collections, lists, strings, text, symbols, sets, bags, and dictionaries.

Exercises

1. Find the number of library references to Array, OrderedCollection, SortedCollection, List, Set, Bag, and Dictionary.
2. All classes understand message allInstances which returns an array with all currently existing instances of the class. As an example, Bag allInstances returns all instances of active instances of Bag currently in the system. Use this message to find the number of instances of Array, OrderedCollection, SortedCollection, List, Set, Bag, Dictionary. The number, of course, depends on the current state of your session.
3. Extend the previous exercise by calculating the smallest, largest, and average size of collection instances.
4. For each of the concrete collection classes listed in this section, give an example of one situation in which the collection would be appropriate.

7.3 Properties shared by all collections

Before presenting individual collection classes in detail, we will now survey their most important shared characteristics, most of them defined in the abstract class `Collection`. They include protocols for creating collections, adding and deleting elements, accessing them, converting one kind of collection into another, and enumeration.

Creation of new collections (class methods)

The most common messages for creating collections are `new` or `new:`. Both are inherited from class `Behavior` which defines most of the shared functionality of all class methods across Smalltalk library. The typical uses of `new` and `new:` are as follows:

Array new: 20	"Creates an instance of Array with 20 uninitialized elements."
Set new	"Creates a Set object with room for a default number of elements (2)."

To create a collection with only a few initially known elements, use one of the `with:` creation messages as in:

Array with: 4 factorial	"Creates an Array with the single element 4 factorial."
Array with: students with: teachers	"Creates a two-element Array with the specified elements."

Predefined `with:` messages have up to four keywords (`with:with:with:with:` is the last one) but you can easily define your own `with:` messages with more keywords if you wish.

Another useful creation message is `withAll: aCollection` which creates a new collection and initializes it to all elements of `aCollection`. As an example,

`OrderedCollection withAll: anArray`

creates an `OrderedCollection` containing all elements of `anArray`. Object `anArray` is not affected.

Several collection classes have their own specialized creation messages in addition to the ones listed above. Moreover, arrays have the distinction that they can be created as *literal arrays* without any creation message. As an example,

`#(34 43 78 -22)`

creates a four-element array containing the four specified numbers. We will say more about this later. Collections are also often created by converting existing collections as explained next.

Converting one kind of collection to another (instance methods)

Almost any kind of collection can be converted into almost any other collection. The process creates a *new collection* containing the elements of the original collection but *the original is not changed*. As we have already seen, conversion messages generally begin with the word `as`.

Conversion is sometimes used just to perform some operation on the receiver. As an example, if `anArray` contains some numbers and we want to sort them, message

`aSortedCollection := #(34 43 78 -22) asSortedCollection`

creates a `SortedCollection` and fills it with the elements of `anArray` sorted in ascending order of magnitude. When you execute

```
| anArray aSortedCollection |
anArray := #(32 96 -34 89 32 45).
aSortedCollection := anArray asSortedCollection
```

you will get SortedCollection (-34 32 32 45 89 96) and anArray remains changed. (This is a very important property of all conversion messages that is frequently overlooked, causing problems that may be difficult to trace.)

To convert the result back to an array, we could use

```
anArray := anArray asSortedCollection asArray.      "Sort and convert back to Array."
```

The combination of the asSortedCollection and asArray thus sorts the array. Another frequently used conversion message is asSet, as in

```
aCollection asSet
```

which creates a Set with elements from aCollection and removes all duplicates. The receiver is not affected.

```
anArray := anArray asSet asArray
```

removes all duplicates from anArray and stores them in some unpredictable order in a new array.

Another reason for performing conversion is that some operation require a collection of a kind different from the receiver's class. As an example, a popup menu expects its labels to be supplied in an sequenceable collection. If the labels are, for example, in the form of a Set, we must convert them as in

```
listOfLabels := aSet asArray
```

Accessing (instance methods)

One of the main distinctions between different collection classes is how their elements are accessed and so Collection leaves accessing of individual elements to concrete collection classes. Its own accessing protocol is limited to *size* and *capacity* corresponding, respectively, to the number of elements present in the collection, and the space available in the collection.

```
aCollection capacity
```

returns the number of *slots* available in aCollection for 'storing' its elements. Note that if the capacity of a collection is 5, the collection does not necessarily contain five elements; this only means that in its present state, the collection has room for five elements (Figure 7.3).

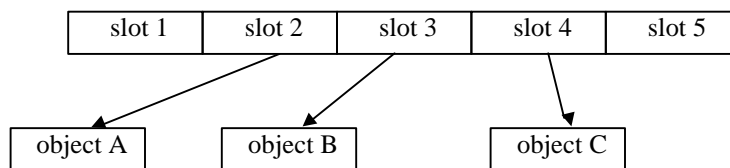


Figure 7.3. A collection with capacity 5 and size 3.

The reason why we put 'storing' in quotes is to emphasize that collections don't really *store* their elements. In other words, the representation of a collection object in computer memory does not *contain* the representation of its elements. It only contains references to these objects, *pointers* to their binary representations - essentially their memory addresses - and each slot is thus only big enough to store a pointer to the corresponding element. This has two important consequences. One is that collections as such require relatively little memory space, and the other is that two or more different collections can share elements. This should not surprise you because *all* Smalltalk objects, not just collections, access their components via pointers. This principle is, however, worth repeating because the potentially large number of elements of collections might lead one to think that collections must occupy a lot of memory space. They don't - but their elements might. We will ignore this technical detail in the future and generally say that a

collection 'contains' or 'stores' its elements and depict collections as if they contained their elements although this is not literally true.

The complement of capacity, the size message, as in

`aCollection size`

tells us how many slots of `aCollection` actually contain pointers to objects, in other words, how many elements the collection really 'has'. For arrays, capacity and size are always the same; for most other collections, capacity and size are different concepts.

Enumeration - doing something with all elements in a collection (instance methods)

One of the most common operations on collections is executing one or more statements with each of its elements. As an example, one might want to print each element in the Transcript, or calculate the square of each element. This is called iteration or enumeration. The most common enumeration messages are illustrated by the following examples in which the block argument is applied to each element of the receiver collection:

```
| aCollection selections rejections squares sum product|
aCollection := #(13 7 89 11 76 4 65 32).
aCollection do: [:element| Transcript cr; show: element printString].
selections := aCollection select: [:element| element > 45].      "Collection of all elements > 45."
rejections := aCollection reject: [:element| element > 45].      "Collection of all elements not > 45."
squares := aCollection collect: [:element| element squared].      "Collection of all element squares."
sum := aCollection inject: 0 into: [:element :tempSum| tempSum + element]. "Sum of all elements."
product := aCollection inject: 1 into: [:element :tempProd| tempProd * element] "Product of all elements."
```

The `do: message` executes the block for each element of the receiver. In our example, it displays each element of `aCollection` in the Transcript. Receiver `aCollection` is not affected in any way. Message `do:` is the most important enumeration message and can be used to implement all other types of enumeration. In fact, message `do:` is the only enumeration message which is left as subclass responsibility and all others are fully defined on the basis of `do:` in class `Collection` (some are redefined in a few subclasses). In spite of the power of `do:`, experienced Smalltalk programmers always use the specialized messages below when appropriate to avoid unnecessary work and possible errors involved in re-inventing the operation.

The next three enumeration messages all *create a new collection* without changing the receiver. As a result, if you don't save the result in a variable or reuse it immediately, the result is immediately lost.

Message `select:` creates a new collection containing only those elements that satisfy the block, in other words, those elements for which the block evaluates to true (Figure 7.4).

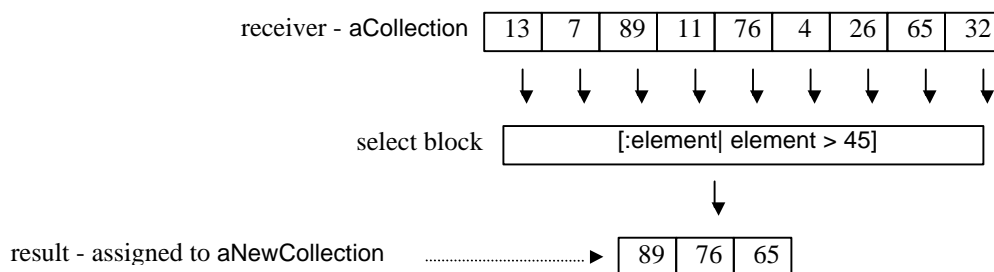


Figure 7.4. `aNewCollection := aCollection select: [:element| element > 45].`

In our example, the result of the simple test is a new collection containing only those elements of `aCollection` that are greater than 45. The tests can be much more complex, as long as the block returns true or false for each element, and the whole expression fails if even one element of `aCollection` cannot execute the block. This is, of course, true for all enumeration messages.

Message reject: is the opposite of select: - it rejects all elements that evaluate the block to true and puts the remaining elements into a new collection. In our case, the new collection will contain all elements of aCollection that are not greater than 45, in other words, all elements smaller or equal to 45.

Message collect: creates a new collection whose size is identical to that of the receiver and whose elements are the objects returned by evaluating the block. In our case,

```
squares := aCollection collect: [:element| element squared]
```

calculates a collection of squares of the original numbers and assigns it to squares.

The inject:into: message uses a block with two arguments. The first argument enumerates over all elements of the collection, the second is the initial value of some calculation performed by the block. In general, the expression in the block takes the element (the first argument) and updates the value of the intermediate result (the second argument). The message returns the last value of the second argument.

Message inject:into: is very powerful but not used much by beginners, perhaps because it is not so easy to understand as other enumerations and because of confusion over which argument is which. To help with the second problem, we suggest that you remember that the first arguments is the receiver's element – and if you read the expression from left to right, the receiver comes first too.

All enumeration messages that create collections, including collect:, select:, and reject:, create a collection of the same *species* as the receiver. In most cases, the species of a class is the class itself (the species of Array is Array, the species of OrderedCollection is OrderedCollection, and so on) because most classes inherit the following definition of species defined in Object:

species

```
"Answer the preferred class for reconstructing the receiver. Species and class are not always the same."  
^self class
```

In a few classes, the most suitable result is not of the same kind as the receiver and species is then redefined, returning a class different from the class of the receiver.

The handling of species is typical for dealing with messages that behave identically for almost all classes with a few exceptions. Such methods are usually defined in class Object or a suitable superclass in a way that reflects the predominant behavior, and inherited. The few classes that require different behavior override the inherited definition.

The definition of enumeration methods is a classical example of the use of inheritance. All of them are defined in terms of do: and only do: is left to subclasses - as noted prominently in Collection's comment. As an example, the definition of collect: in Collection is as follows:

collect: aBlock

```
"Evaluate aBlock with each of the values of the receiver as the argument. Collect the resulting values into  
a collection that is like the receiver. Answer the new collection."  
| newCollection |  
newCollection := self species new.  
self do: [:each | newCollection add: (aBlock value: each)].  
^newCollection
```

Inheriting this definition by all collections, provides a great saving and consistency, and only a few classes redefine collect: because their internal structure allows them to perform it more efficiently.

Testing (instance methods)

The testing protocol of collections allows checking whether a collection contains objects satisfying a block, specific objects, or any objects at all. It includes the following messages:

aCollection isEmpty	"Returns true or false."
aCollection includes: 'Air Canada'	"Returns true or false."
aCollection contains: [:element] element > 7]	"Returns true or false."
aCollection occurrencesOf: 'abc'	"Returns an integer number."

As an example of the implementation of these messages, the definition of isEmpty is simply

isEmpty

```
^self size = 0
```

Message includes: returns true if at least one of the elements of aCollection is equal to the string 'Air Canada'; it returns false otherwise.

Message contains: returns true if at least one element of aCollection satisfies the *block argument*; it returns false otherwise. Just as in enumeration, it is important to remember that during its execution, contains: may have to execute the block argument for each element of aCollection. Consequently, if even a single element of the collection cannot execute the block, the message may fail. As an example, if all but one of the elements of aCollection in our example are numbers, the message may generate an error because a string cannot compare itself with a number. Ignoring this principle is another common programming mistake.

Finally, occurrencesOf: counts how many occurrences of the collection are equal to the argument. Its definition uses do: as follows:

occurrencesOf: anObject

"Answer how many of the receiver's elements are equal to anObject."

```
| tally |  
tally := 0.  
self do: [:each | anObject = each ifTrue: [tally := tally + 1]].  
^tally
```

Adding elements (instance methods)

If a collection can grow - and most collections can - it understands messages add: and addAll:.

Message add: anObject adds anObject in a way consistent with the nature of the collection (ordered or unordered) and 'grows' the collection by increasing its capacity if necessary. Growing can be a very time-consuming because it first creates a new larger collection, and then copies all elements of the original collection into it. It then causes the new collection to 'become' the original collection, changing all existing references to the old version to refer to the new version. (Message become: from class Object performs this operation.) This overhead can be very significant and the only way to avoid it is to try to predict the maximum capacity that the collection will ever need and create it initially with this (or larger) capacity using, for example, the new: message.

Creating a collection whose capacity may never be fully utilized may appear to be wasteful but since a collection is just a bunch of pointers, its memory requirements are small and probably worth the run time saved in growing. The methods that make growing possible also use this strategy - when a collection's capacity must be increased, it is not increased by adding one slot but by adding several slots so that consecutive add: or addAll: messages don't have to grow the collection again.

Message addAll: aCollection copies *each element* from aCollection into the receiver. As an example, if aCollection contains 25 elements, the receiver collection will gain 25 new elements, growing in the process if necessary. (If the receiver is a Set, all duplicates will, of course, be ignored.)

Both add: and addAll: are frequently used and their behavior is slightly but critically different. As an example, consider the following code and the illustration of its effect in Figure 7.5:

collection1 add: collection2.	"Adds collection2 to collection1 as a single objects."
collection1 addAll: collection2	"Adds all elements to collection1 one by one."

Message add: anObject adds anObject to the receiver collection as a single new element, whereas addAll: anObject assumes that anObject is a collection, 'takes it apart', and adds each element to the

receiver individually. In both cases, the receiver must be able to grow which means that `add:` and `addAll:` cannot be used with arrays.

In both `add:` and `addAll:` in our example, `collection1` and `collection2` share elements and if the program later modifies any of these elements, both collections will be affected. If, however, one collection *replaces* one of the elements with a new object, the corresponding pointer will point to the new object but the other collection will not be affected because it still points to the original object. We will clarify this point later.

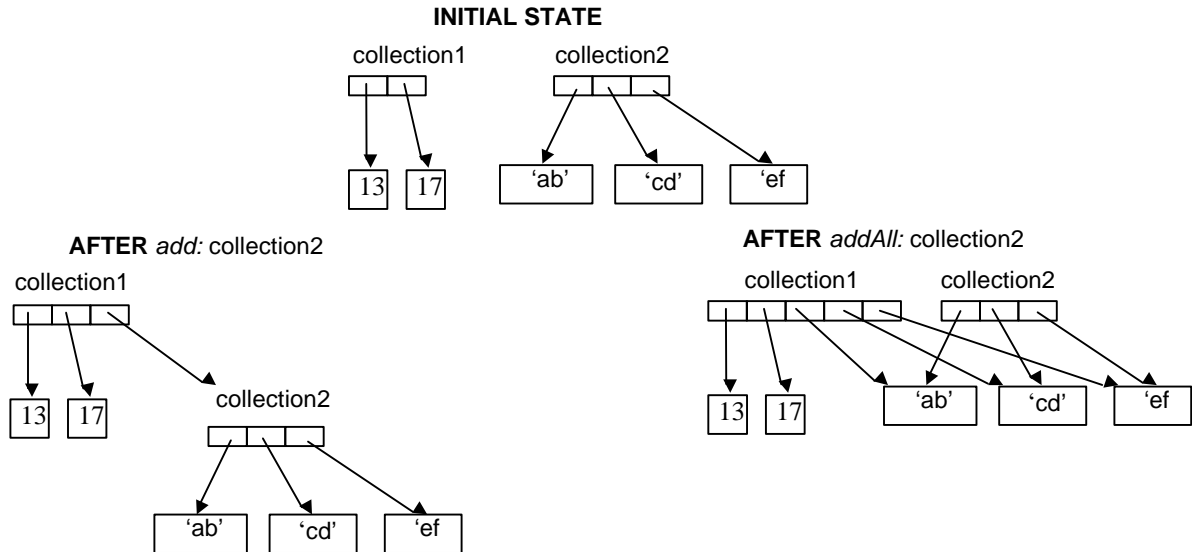


Figure 7.5. Effect of `collection1 add: collection2` (left) and `collection1 addAll: collection2` (right).

Class `Collection` leaves `add:` as subclass responsibility because the exact place at which the element is added depends on the nature of the receiver. Method `addAll:`, on the other hand, is fully defined in `Collection` using `add:` and subclasses only need to define `add:`. The shared definition of `addAll:` uses enumeration:

addAll: aCollection

```
"Include all the elements of aCollection as the receiver's elements. Answer aCollection."
aCollection do: [:each | self add: each].
^aCollection
```

Unlike other methods (such as conversion) that make a copy of the receiver and work on the copy without changing the receiver, `add:` and `addAll:` change the receiver. This is natural but worth mentioning and you can test it on the following code:

```
| orderedCollection |
orderedCollection := OrderedCollection with: 'abc' with: 'xyz'.    "OrderedCollection ('abc' 'xyz')."
orderedCollection add: 'uvw'.    "Changes orderedCollection but returns the 'uvw' object"
orderedCollection    "Returns OrderedCollection ('abc' 'xyz' 'uvw')"
```

A somewhat unexpected and confusing feature of all `add` (and `remove`) messages is that they *return the argument* rather than the changed receiver. As an example,

```
orderedCollection add: 'uvw'
```

changes `orderedCollection` but returns `'uvw'` rather than the modified `orderedCollection` as one might expect. Failing to realize this is a very frequent cause of errors.

Removing elements (instance methods)

Removing objects from a collection is subject to similar restrictions as adding because collections that cannot grow don't know how to remove elements either. As an example, you can *change* the elements of an array but you cannot *remove* them. Like *add*, *remove* messages also return the argument rather than the receiver as one might expect.

If you are certain that the object that you want to remove is in the collection, use *remove:* as in

```
aCollection remove: 45    "Returns 45."
```

or in

```
aCollection remove: aStudent
```

If the object is not in the collection, this message will raise an error. Consequently, if you are not sure, use the alternative *remove:ifAbsent:* as in

```
aCollection remove: 45 ifAbsent: []    "Do nothing if the element is not in aCollection."
```

or in

```
aCollection remove: aStudent ifAbsent: [Dialog warn: aStudent printString, ' is not in the collection']
```

The first expression will not do anything if the collection does not contain 45, the second will display a warning if *aStudent* is not in the collection. These two styles of *remove:* - one providing an alternative block, and one providing an unprotected operation - are typical for collection messages. Since *remove:* is a special case of *remove:ifAbsent:* it is defined as

remove: oldObject

"Remove oldObject as one of the receiver's elements. Answer oldObject unless no element is equal to oldObject, in which case, provide an error notification. "

```
^self remove: oldObject ifAbsent: [self notFoundError]
```

where *notFoundError* opens an exception window. Method *remove:ifAbsent:* itself is left as subclass responsibility.

Example: A new enumeration method

There are occasions when one needs to perform an operation on each element of a collection except a particular one. A method dealing with this problem might be useful for any kind of collection and will thus add it to the enumeration protocol of the abstract class *Collection* and all other collections will inherit it. In doing this, we must be sure to design the method so that all collections will be able to execute it.

Our method, called *do:exceptFor:*, first constructs a new collection by removing the special element from it, and then sends *do:* to the result to enumerate over all remaining elements:

do: aBlock exceptFor: anObject

"Evaluate aBlock with each of the receiver's elements except anObject."

```
| collection |  
collection := self reject: [:el | el = anObject]].  
collection do: aBlock
```

We tested this approach by printing all elements of a literal array of numbers that are not equal to 3. We used arrays of the following kinds: An array that does not contain 3, an array that contains exactly one element equal to 3, and an array that contains more than one copy of three. As an example, the last of the three tests was

```
 #(1 2 3 4 5 3 4 5 3 4 5) do: [:el| Transcript cr; show: el printString] exceptFor: 3
```

We found that all tests worked but we should, of course, test the method for other kinds of collections as well. This is left as an exercise but for now, we will generalize our method somewhat: In some situations, we might want to execute selective enumeration on the basis of a test rather than by specifying the element explicitly. As an example, we might want to do something with all elements that are greater than 3.

In this problem, we must allow the second argument to be either a block or any other object, and treat blocks in a special way. Our approach will be the same as before but the construction of the collection will be performed differently if the second argument is some kind of a block. In that case, we will simply reject all elements of the receiver that satisfy the block. The whole solution is as follows:

do: aBlock exceptFor: blockOrAny

"Evaluate aBlock with each of the receiver's elements except those implied by blockOrAny."

```
| collection |
collection := (blockOrAny isKindOf: BlockClosure)
               ifTrue: [self reject: [:el | blockOrAny value: el]]
               ifFalse: [self reject: [:el | el = blockOrAny]].
collection do: aBlock
```

where message `isKindOf:` checks whether the receiver is an instance of the argument class or its subclass. We tested that our previous tests still work and then added tests of the following kind:

```
 #(1 2 3 4 5 3 4 5 3 4 5) do: [:el| Transcript cr; show: el printString] exceptFor: [:el| el > 3]
```

We found that all tests work as expected.

After this general introduction to collections, we will now examine the essential collections in more detail and illustrate them on examples. In the rest of this chapter, we will present sequenceable collections, all of them subclasses of the abstract class `SequenceableCollection`. `SequenceableCollection` implements some of the shared protocols, among them the all important `do:` which it defines as follows:

do: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument."

```
1 to: self size do: [:i | aBlock value: (self at: i)]
```

As you can see, the definition is based on the `to:do:` message defined in number classes and uses the essence of `SequenceableCollection` - the fact that its elements are accessible by consecutive integer indices starting at 1.

Main lessons learned:

- Abstract class `Collection` defines many behaviors shared by all collections.
- The main protocols of collections include creation, accessing, testing, adding and removing elements, conversion to other kinds of collections, and enumeration (iteration over collection elements).
- Conversion is often used to perform operations such as elimination of duplicates or sorting.
- When using collection methods, make sure to understand what kind of object the method returns and whether it changes the original or returns a modified copy while leaving the original unchanged.
- Enumeration is among the most frequent collection operations and good Smalltalk programmers always use the specialized enumeration messages when appropriate.
- Messages that add or remove elements always return the argument and modify the receiver.

Exercises

1. Test conversion messages `asSet`, `asSortedCollection`, `asOrderedCollection` on literal arrays `#(75 13 254 41 -65 75 75)` and `#('word' 'symbols' 'digits' 'letters' 'letters')` and examine the result and the effect on the receiver.
2. Print the elements of the arrays in the previous exercises in the Transcript in reverse order. (Hint: Check out the enumeration protocol of `SequenceableCollection`.)
3. Execute
| anArray |
anArray := #(32 96 -34 89 32 45).
anArray := anArray asSet asArray
and comment on the result (type of returned collection, its size, and order of elements).
4. Find how collections grow (Hint: Read the definition of `add:`).
5. Classes `SequenceableCollection` and `ArrayedCollection` are abstract. Write their brief summaries with a description of one important method from each of three selected instance protocols.
6. Browse class `Collection`, read the definitions of its enumeration messages, and examine their use by looking at three existing references. Write a summary including one example of each.
7. Find two `Collection` methods redefined in some of its subclasses and describe the differences.
8. Method `add:` in `Collection` should not be used with `Array` receivers. What will happen if you do send `add:` to an instance of `Array`?
9. Find two other examples of the use of the mechanism discovered in the previous exercise. Find definitions of `new` and `new:`. (See also Appendix 1).
10. Under what conditions contains: `will/will not` raise an error?
11. We stated that messages that behave identically in most classes are usually defined in class `Object` in a way that reflects the predominant behavior, and the few classes that require different behavior override the inherited definition. Browse definitions of `species`, `isNil`, and `isSymbol` as examples of this technique.
12. What is the difference between
aCollection remove: 45 ifAbsent: [self]
and
aCollection remove: 45 ifAbsent: [^self]
13. `Interval` is a collection frequently used to represent an arithmetic progression of numbers defined by start, end, and step values, and its main use is for certain types of enumeration. Find `Interval` in the browser, examine its uses, and write a short summary of your findings.

7.4 Arrays

Among the numerous collection classes, arrays are perhaps conceptually the simplest. They are also among the most frequently used and we will thus start our coverage of collections with them.

Arrays are fixed-size sequenceable collections and their elements are numbered by successive integer indices starting from 1. Unlike some other sequenceable collections, elements of arrays don't change their position during the array's lifetime. Arrays are accessed mostly by enumeration and less frequently by index. Since we have already presented enumeration messages which all apply almost identically to all collections, we will start with element accessing.

Accessing an array element

To obtain the value of the element at a particular index, send the `at: index` message as in

anArray at: 7 "Returns element at index 7. Fails if anArray has fewer than 7 elements."

To replace the element at a known index with a new object, send the `at:put:` message. As an example,

anArray at: 7 put: 0.4 sin "Replaces element at index 7 with 0.4 sin. Fails if anArray size < 7."

replaces the seventh element with the object 0.4 sin.

Just like add and remove messages, the at:put: message changes the receiver but *returns the second argument*. As a consequence, the statement

```
anArray := anArray at: 3 put: 5 factorial
```

changes the value of variable anArray to 120 - probably not what was desired (Figure 7.6). If all we wanted was to change the array, we should have written simply

```
anArray at: 3 put: 5 factorial
```

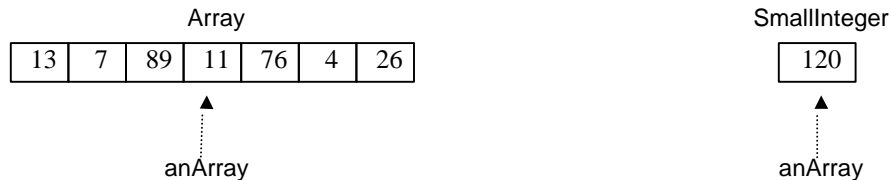


Figure 7.6. Value of anArray before (left) and after (right) executing `anArray := anArray at: 3 put: 5 factorial`.

An important aspect of array accessing is that Smalltalk always checks whether the index is within the bounds of the interval `<1, size>`. If the index is out of bounds, the attempted access produces an Exception window. This automatic checking adds overhead time and some people think that bound checking should be optional. The great majority of Smalltalk programmers consider the security provided by bounds checking more important than speed of access and eliminating bounds checking is not an option.

Creating new arrays

Arrays are usually created either as literals, or by a creation message such as `new:` or `with:`, or by conversion. We will now illustrate all these possibilities.

The easiest way to create an array is as a *literal* as in

<code>array := #(1 3 5 7).</code>	"Four-element array containing four numbers."
<code>array := #('string 1' 'string 2').</code>	"Two-element array containing two strings."
<code>array := #(\$a \$e \$i \$o \$u)</code>	"Five-element array of characters."
<code>array := #('string' #symbol).</code>	"Two-element array with a string and a symbol."
<code>array := #(#symbol1 #symbol2 #('abc' 4))</code>	"Three-element array with two symbols and a sub-array."

Executing

```
array at: 3
```

returns 5 for the first array, fails with the second and fourth, returns array `#('abc' 4)` with the fifth, and character `$i` for the third. If we wanted to obtain the second element of the *nested array* in the fifth example, we would have to extract the sub-array first and then access its second element as in

```
(array at: 3) at: 2
```

"Returns 4."

The limitation of literal arrays is that they can only be created with literal components, in other words, only with the following *literal* objects: nil, true, false, numbers (other than Fraction), characters, strings, symbols, and literal arrays. The following statements thus will not produce what you might expect:

<code>array1 := #(3 factorial 5 factorial 7 factorial).</code>	"Same as <code>#(3 #factorial #5 #factorial #7 #factorial)</code> ."
<code>array2 := #(3/4 (5/6 7/8)).</code>	"Same as <code>#(3 #/ 4 #(5 #/ 6 7 #/ 8))</code> ."
<code>array3 := #(student1 student2)</code>	"Same as <code>#(#student1 #student2)</code> ."

When an array cannot be created as a literal array, the following methods are the most common:

1. Use the new: message and assign elements by enumeration or by accessing individual indices.
2. Use one of the with: messages.
3. Obtain the array from another collection by conversion.
4. Calculate the array from another array by enumeration.

Method 1: Using new: to create an uninitialized array and calculating individual elements

Let's construct an array containing the squares of ten consecutive numbers starting with 37. Since the values of the elements must be calculated, we create an array of size 10 and use enumeration over the index as follows:

```
array := Array new: 10.           "Create an array of size 10 with all elements initially nil."
1 to: 10 do
    [:index| array at: index put: (index + 36) squared]    "Calculate elements."
```

As another example, the following code fragment shows how we could prompt the user to fill an array of size 5 with numbers. We again use the index as the basis of the solution:

```
| array |
array := Array new: 5.
1 to: 5 do:
    [:index| | string |
        string := Dialog request: 'Enter a number'.
        array at: index put: string asNumber]
```

A simpler solution that eliminates the temporary block variable is

```
| array |
array := Array new: 5.
1 to: 5 do:
    [:index| array at: index put: (Dialog request: 'Enter a number ') asNumber]
```

Sometimes the elements of an array cannot be obtained as neatly as this. As an example, assume that we have a class StudentRecord with instance variables studentID, firstName, lastName, middleInitial, streetAddress, city, province, postalCode, coursesTaken, registrationYear, and degree. Assume that our application requires an array containing studentID, firstName, lastName, province, and postalCode, and that StudentRecord provides accessing methods to access these components. In this case, we must create the array as follows:

```
selectedComponents := Array new: 5.    "Returns #(nil nil nil nil nil)."
selectedComponents at: 1 put: aStudentRecord studentID.
selectedComponents at: 2 put: aStudentRecord firstName.
selectedComponents at: 3 put: aStudentRecord lastName.
selectedComponents at: 4 put: aStudentRecord province.
selectedComponents at: 5 put: aStudentRecord postalCode
```

Finally, note that Array also understands the new message but this message is almost useless because it creates an array with size 0.

Method 2: Creating an array using one of the with: messages

This approach is useful when we need to create an array with up to four known elements and cannot use the literal form, as in

```
array1 := Array with: 5 factorial with: aNumber with: 3
array2 := Array with: Student new with: Address new with: Marks new
```

Neither of these arrays could be constructed as a literal array because all their elements are not literals.

Although the maximum number of `with:` keywords in the `Array` protocol is four (message `with:with:with:with:`) you could define a method with more arguments by creating an uninitialized array and then assigning the individual arguments as in

with: firstObject with: secondObject with: thirdObject with: fourthObject with: fifthObject

```
| newCollection |
newCollection := self new: 5.
newCollection at: 1 put: firstObject.
newCollection at: 2 put: secondObject.
newCollection at: 3 put: thirdObject.
newCollection at: 4 put: fourthObject.
newCollection at: 4 put: fourthObject.
newCollection at: 5 put: fifthObject.
^newCollection
```

This style is used by the predefined `with` messages.

Method 3: Creating an array by converting another type of collection

It occasionally happens that we need an array but don't know how many elements it will have. In this case, we start with a suitable type of collection to collect the elements, and convert it to an array as in

```
"Create variable size collection shoppingList."
```

```
...
```

```
"Convert it to an Array."
```

```
shoppingList := shoppingList asArray
```

Note that simply sending the conversion message would not give the desired result

```
"Create collection shoppingList"
```

```
...
```

```
"Convert it to an Array"
```

```
shoppingList asArray
```

because the last statement creates an array from collection `shoppingList` but does not change the value of variable `shoppingList` itself. The creating array is thrown away because it is not referenced by any object.

Method 4: Creating an array by enumeration

Creating a collection by enumeration is very common. As an example, the following code fragment uses an array of strings to create another array called `shortStrings` containing only those strings that contain at most 15 characters:

```
| strings shortStrings |
strings := #( 'short string 1' 'in our context, this is a long string' 'short string 2').
"Get a new array by selection."
shortStrings := strings select: [:string| string size <= 15].
etc.
```

Note that we did *not* have to convert the result to an array even though its size is different from that of the original, because the species of receiver `strings` is `Array`.

As another example, assume that we want to calculate an array of cosines of arguments stored in another array. To do this, we use `collect:` as follows:

```
| argArray cosArray |
argArray := #(0.1 0.15 0.25 0.4 0.45 0.5).
```

cosArray := argArray collect: [:arg| arg cos].
etc.

Note that we could not use to:by:do: because the arguments stored in argArray are not regularly spaced.

Main lessons learned:

- An array is a fixed-size sequenceable collection whose elements are accessed by consecutive integer indices numbered from 1.
- Arrays, like most other Smalltalk collections, allow any objects to be their elements.
- Arrays are usually created as literals or by conversion from other types of collections, using new: followed by calculation of individual elements, or using one of the with: messages.
- When an array is created, it is assigned a fixed size and all its elements are initialized to nil.
- Each array access checks whether the index is within the array's index bounds.
- The main advantage of arrays is their efficient accessing.

Exercises

1. Array's definition of collect: is not inherited from Collection. Find where it comes from and what saving it provides compared to the Collection definition.
3. Inspect #(3/4 5/6 7/8).
4. What happens when you execute #(1 2 3 4) at: 7?
5. Since any Smalltalk programs can fail and produce an Exception window, what is so bad about failures that could occur if Smalltalk did not have automatic bounds checking?
6. One of the following expressions succeeds and the other fails. Explain why.
#(13 53 21 87 9 'abc') contains: [:n| n>200]
a. #(13 53 21 87 9 'abc') contains: [:n| n>20]
7. Write a code fragment to print
 - a. successive elements of an array separated by carriage returns (use do:separatedBy:)
 - b. elements of an array with their indices (use keysAndValuesDo:)
 - c. elements of an array in reverse order (use reverseDo:)
8. When you try to print a large array such as Smalltalk classNames asArray, you will only get some of its elements. Which method controls this behavior?
9. Use the inject:into: message to solve the following tasks where appropriate.
 - a. Find the sum of all elements of an array of numbers.
 - b. Find the product of all elements of an array of numbers.
 - c. Repeat the previous problem without assuming that all elements of the array are numbers.
 - d. Find the product of all non-zero numbers in an array whose elements are all numbers.
 - e. Find the largest number in an array whose elements are all numbers.
 - f. Find the smallest rectangle containing all rectangles in an array of rectangles. (Hint: Check the protocols of Rectangle and create rectangles using originPoint corner: cornerPoint as in 1@1 corner: 10@15.)
 - g. Find the largest rectangle contained in all rectangles in an array of rectangles.
10. Calculate the following collections using enumeration messages:
 - a. The string containing all vowels in 'abcdefg'. (Hint: Use message isVowel.)
 - b. The string containing all characters that are not vowels in 'abcdefg'.
 - c. The string of characters in 'abcdefg' shifted by one position – in other words, 'bcdefgh'. (Hint: Convert a character to its integer code using asInteger, then increment the code, and convert to Character again.)
 - d. String entered by the user, encoded by shifting characters as in the previous exercise.
 - e. All numbers between 1 and 5000 that are divisible by 7 and 11 but not by 13. (Hint: The species of an Interval is an Array and an expression such as (1 to: 5000) select: aBlock thus produces an array.)
11. Calculate the sum of 10,000 numbers using do: and then using inject:into: and compare the speed.

12. Perform the following tasks using enumeration messages:
 - a. Create an array of three rectangles and use the current window to display their outlines. (Hint: To display the circumference of aRectangle in the current window, use expression aRectangle displayStrokedOn: Window currentWindow component graphicsContext. Message displayFilledOn: displays the rectangle filled. We will talk about graphics in detail in Chapter 12.)
 - b. Ask user for the side s of a square and for a displacement d, construct an array containing five squares with the specified side whose origins are displaced by d@d, and display the filled squares in the current window. (Hint: See class Rectangle for rectangle operations, and the previous exercise for information about display.)

7.5 Examples of uses of arrays

Smalltalk programmers use arrays extensively but not as universally as programmers in some other languages. One reason for this is that if you really need a collection, Smalltalk provides many other types of collections which are often more suitable because they can grow, because they sort their elements, eliminate duplication, and so on. And if you need an object whose components have identifiable roles, the only correct solution in most cases is to define a class with named instance variables.

When you browse the Smalltalk library, you will find many references to arrays. Most of them use an array because they require enumeration and enumeration is particularly efficient with arrays.

After this introduction, we will now do three more examples showing the use of arrays.

Example 1. Multiple choice dialogs

A typical use of arrays is for creating menus and multiple choice dialogs. As an example, the following statement produces the dialog window in Figure 7.7 and returns the selected ice cream flavor if the user clicks *OK*, or an empty string if the user clicks the *Cancel* button.

```
Dialog choose: 'Which kind of ice cream do you prefer?'  
fromList: #('vanilla' 'strawberry' 'raspberry' 'banana' 'pecan' 'pistachio')  
values: #('vanilla' 'strawberry' 'raspberry' 'banana' 'pecan' 'pistachio')  
lines: 6  
cancel: ['']
```

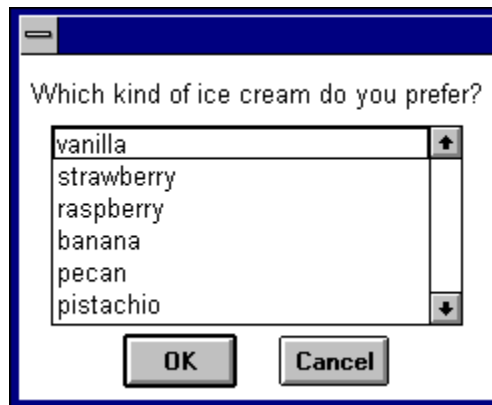


Figure 7.7. Multiple choice dialog produced with literal arrays.

The message `choose:fromList:values:lines:cancel:` is one of several very useful multiple choice dialogs in class `Dialog`. The argument of `choose:` is the prompt line, the argument of `fromList:` is an array of displayed strings, the `values:` argument is an array of objects that are returned when the corresponding label is selected (its elements may be anything you want but we chose strings identical to the labels), `lines:`

specifies how many lines should be displayed at a time (the rest can be scrolled), and the last argument is a block that will be executed if the user clicks *Cancel* (returns an empty string in our example).

Example 2. Using arrays for mapping

A interesting use of arrays is for converting one collection of objects into another. Assume, for example, that we want to convert some text into unreadable strings. In a very simple scheme, each letter in the message could be replaced by some other fixed letter, preferably without any easily discernible pattern. As an example, letter \$a could be replaced by letter \$f, letter \$b by letter \$z, \$c by \$v, \$d by \$r, and so on. With this scheme, a string such as 'acdc' would be encrypted into 'fzrz'. This task can be nicely implemented by storing the encryption scheme in a literal array such as

```
encryption := #($f $z $v $r etc.)
```

and extracting the encoding from the array, using the original character's position in the alphabet to calculate the index. As an example, \$a is the first letter of the alphabet and so we use the first letter in the encryption array to encrypt it. Character \$b is the second letter in the alphabet and its encryption character is the second character in the encryption array, and so on (Figure 7.8).

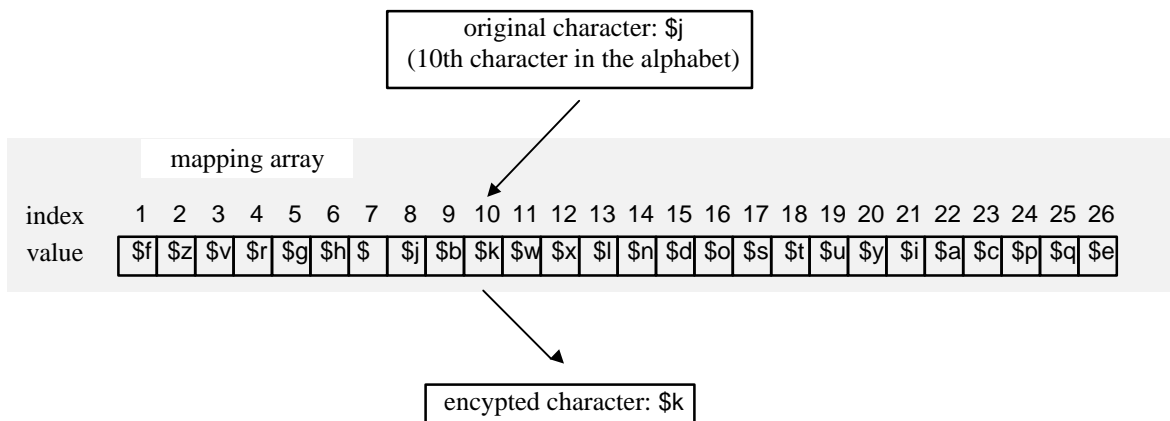


Figure 7.8. Encrypting secret message using a mapping array.

Since each character has a numeric code (the ASCII code¹) which is related to its position in the alphabet (the code of \$a is 97, the code of \$b is 98, and so on), and since the code of a character can be obtained by sending message asInteger, the formula for finding the index of a character is

(character asInteger) - (\$a asInteger) + 1 "Returns 1 for \$a, 2, for \$b, and so on."

We will now write a code fragment to ask the user for a string, print it, encrypt it using the mapping in Figure 7.8, and print the encrypted string. As an example, the program will encode the string 'a multi-word string' into 'lixyb-cdtr uytbnm'. The basis of the program is the use of collect: to calculate the encrypted string from the original string:

```
| encrypted encryption original shift |
"Initialize an encryption array, making sure that each letter appears exactly once."
encryption := #($f $z $v $r $g $h $m $j $b $k $w $x $l $n $d $o $s $t $u $y $i $a $c $p $q $e).
"Pre-calculate and cache index shift to avoid the need to recalculate it for each character."
shift := $a asInteger - 1.
"Get string from user."
original := Dialog request: 'Enter a string' initialAnswer: "".
```

¹ ASCII (American Standard Code for Information Interchange) is a widely used set of codes of printable characters and control characters such as Line Feed and Escape.

```
"Display original."  
Transcript clear; show: original; cr.  
"Encrypt and display."  
encrypted := original collect: [:char| encryption at: char asInteger - shift].  
Transcript show: encrypted
```

Unfortunately, if the character is not a lowercase letter, the index calculated in the collect: block falls outside of the limits of the array and causes an exception. We leave it to you to correct this shortcoming.

Example 3. Creating the encrypting array automatically

Instead of having to create an encrypting array by hand as we did in Example 2, we can create it randomly as follows: Start with an array containing all 26 letters of the alphabet. Create a random number between 1 and 26 and insert the corresponding letter at the start of the encryption array; then remove the letter from the original array. In the next iteration, create a random number between 1 and 25, extract the corresponding letter, put it at index 2 of the encryption array, and so on. The principle is as follows:

```
"Create a string with all letters as a start for the mapping array."  
alphabet := 'abcdefghijklmnopqrstuvwxyz'.  
"Create a random number generator, create a mapping array with 26 elements, initialize index."  
"Calculate encryption array."  
26 to: 1 by: -1 do: whileTrue:  
    [:max| "Get a random number between 1 and max, extract corresponding character from  
    alphabet, put it in the mapping array, remove it from alphabet].
```

This is all relatively simple but we must find how to remove an element from a String, in other words, how to make a copy of the original without the removed character. Browsing the collection classes (a string is a collection of characters), we find that there is a useful message called copyWithout: anElement defined in SequenceableCollection and therefore applicable to strings. When we test it with

```
'abc' copyWithout: 'b'
```

it does not work because the elements of strings are characters. When we try

```
'abc' copyWithout: $b
```

we get the correct result 'ac'. We can now write our code for creating the encoding array as follows:

```
| rg map alphabet |  
"Create alphabet string and random generator, initialize index for accessing mapping array."  
alphabet := 'abcdefghijklmnopqrstuvwxyz'.  
rg := Random new.  
map := Array new: 26.  
1 to: 26 do: [:index | | ch |  
    "Extract characters at random locations, deleting the selected character from the alphabet each time."  
    ch := alphabet at: (rg next * alphabet size) truncated + 1.  
    map at: index put: ch.  
    alphabet copyWithout: ch].  
"Print result in Transcript to see if it works."  
Transcript show: map printString
```

The code works correctly.

Example 4. Using arrays to execute a list of messages

A clever and common use of arrays is for execution of one of several alternative messages in a list or all the messages in a list. We will use this idea to print the values of sin, cos, ln, exp, and sqrt for x = 1 to 10 in increments of 0.5 in the Transcript.

Solution: We will use enumeration over `##sin #cos #ln #exp #sqrt` and the powerful `perform: message`. The `perform: message` requires a `Symbol` argument, treats it as a method, and executes it. As an example,

```
5 perform: #factorial
```

has the same effect as

```
5 factorial
```

We would, of course, never use `perform:` if we knew exactly which message we want to send (as in `5 perform: #factorial`) but if the message may be any one of several messages, as in our example, or is supplied as a variable or an argument, `perform:` is the only way to go. With the `perform: message`, the solution of our problem is simply

```
1 to: 10 by: 1/2 do: [:x |
    Transcript cr; show: x printString.
    ##sin #cos #ln #exp #sqrt do:
        [:message || value |
            value := x perform: message.
            Transcript tab; show: value printString]]
```

Example 5. Inappropriate use of arrays

As an example of an *inappropriate* use of arrays, assume that we need an object containing an address with components including country, state, city, street name, and postal code. Each of these components has an identifiable role and we should define a class with named instance variables to hold the individual components. If we used an array to represent this object and put the city name, for example, into the fourth element, we would have to access city names as in

```
address at: 4 put: 'Halifax'
```

which is not intuitive because the index in expression `at: 4 put: 'Halifax'` does not provide any hint that we are dealing with a `nameOfCity` object. This is dangerous because it makes it too easy to use the wrong index with the result of storing the name of the city in the wrong element. Similarly, accessing a component of an array by an expression such as

```
address at: 4
```

does not make any sense and requires that you remember the nature of element 4. A very unpleasant possibility is that somebody (maybe even you) changes the usage of the array so that city name is now the third rather than the fourth element, and does not change all code that depends on it. Our code will then give an incorrect result.

For all these reasons, arrays should not be employed in this way. And if they are, access to their elements should at least be explicit, for example by special accessing methods such as

city

```
"Return the value of city."
^array at: 4
```

city: aString

```
"Change the value of city"
array at: 4 put: aString
```

With these methods, we can replace the ugly

```
address at: 4 put: 'Halifax'
```

with

address city: 'Halifax'

which is both more readable and safer. If somebody changes the meaning of individual elements, he or she only needs to change the definition of accessing messages and all code using the array will continue working as before. If the elements were accessed by index, *all* uses of the index would have to be found and corrected. Nevertheless, as we already mentioned, the best solution usually is to gather the components as named instance variables in a new class called Address.

Main lessons learned:

- Like other collections, arrays are used to hold and process nameless elements. Their advantages over other collections are greater efficiency of access and smaller memory requirements.
- Arrays are not used as much in Smalltalk as in other languages because Smalltalk provides many other types of collections and because a class with named components is often preferable.
- Literal arrays are often used for multiple choice dialogs and popup menus.
- Arrays are useful to map one set of objects into another.

Exercises

1. Modify the multiple choice dialog example so that if the user clicks *Cancel*, the program opens a dialog window showing: We regret that you don't like our ice creams. To display the apostrophe, repeat it as follows: 'We regret that you don't like our ice creams.'
2. Check out other multiple choice dialogs. Use choose: fromList: values: buttons: values: lines: cancel: to create a multiple choice dialog as in Figure 7.9. Making a selection in the list should return the same string as the label, clicking 'Cancel' should return nil, clicking 'Copy all' should return #copyAll, and clicking 'Delete all' should return #deleteAll.
3. Combine the encryption program and random generation of encryption and add code to decrypt the encrypted string.
4. The idea of mapping with a mapping array looks like a potentially useful operation. Define a class called Mapper that will encrypt and decrypt a string. Decide the details of desired behaviors and implementation and use your class to reimplement the encrypting example.

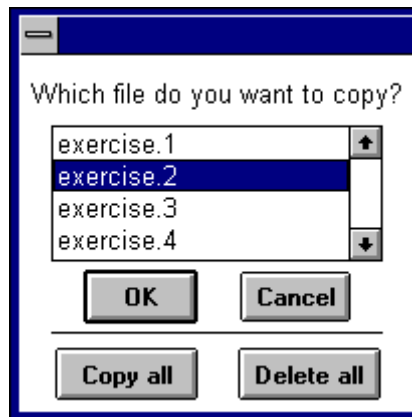


Figure 7.9. Desired user interface for Exercise 2.

7.6 Two-dimensional arrays - tables and matrices

A table is a collection of elements accessible by row and column numbers. It is thus closely related to an array and most programming languages would call it a two-dimensional array; mathematicians would call it a two-dimensional matrix. More generally, an n -dimensional array or matrix is a collection whose elements can be accessed by a set of n indices. An array is thus a one-dimensional matrix, and a table is a two-dimensional matrix.

Two-dimensional matrices are very useful because they capture the substance of a table widget and because they have applications in two-dimensional geometry. While the VisualWorks library does not include a class implementing general n -dimensional matrices, it does have a class called `TwoDList` which implements two-dimensional arrays and uses it as the value holder for the table widget. In this section, we will present class `TwoDList` and show how it could be extended to include mathematical behavior of matrices; the use of `TwoDList` in the table widget will be illustrated in the next section.

Class TwoDList

Class `TwoDList` is a subclass of `ArrayedCollection` and this means that its size is fixed. Its instance variables are `dependents`, `collection`, `rows`, `columns`, and `transposed`. Variable `dependents` makes it possible to use `TwoDList` as a table widget model which is its intended function. Dependency is implemented so that the change of any of its elements sends an update message to all dependents. Variable `collection` holds the elements of the table in a linear array. Variables `rows` and `columns` hold the number of rows and columns of the table, and variable `transposed` holds true or false depending on whether the table elements should be viewed in their original arrangement or transposed - with rows and columns interchanged.

A table is an interesting example of information hiding because the concept can be implemented in at least four ways and the user does not need to know which one is used (Figure 7.10). One possibility is to implement a table as a one-dimensional array of elements and access it by converting row and column indices into a single linear index. (*VisualWorks uses the first index as column number and the second as row number, treating horizontal axis as the x coordinate of a point and vertical axis as y .* This is the opposite of the usual mathematical interpretation.) This can be done in two ways: row-wise (all elements of row 1 followed by all elements of row 2, and so on), or column-wise (all elements of column 1 followed by all elements of column 2, and so on). Another possible implementation is as an array whose elements are rows which are themselves arrays or as an array whose elements are columns which are arrays. As indicated below, `TwoDList` uses one-dimensional row-wise storage.

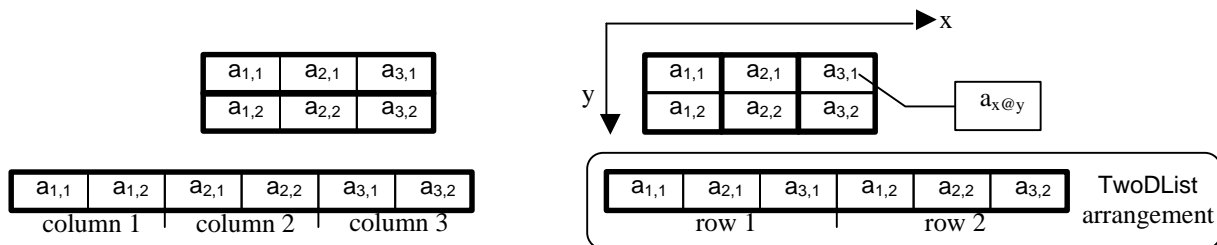


Figure 7.10. Tables can be stored as arrays of columns or rows (top) or as one-dimensional arrays (bottom).
 Class `TwoDList` uses the bottom right arrangement.

Creating a TwoDList

The two main `TwoDList` creation messages are `on: aCollection columns: c rows: r` and `columns: c rows: r`. The first creates a `TwoDList` with elements initialized to `aCollection`, the second creates an uninitialized `TwoDList` with specified shape and size. As an example,

`TwoDList on: #(12 14 16 21 42 24) columns: 3 rows: 2`

returns `TwoDList (12 14 16 21 42 24)` with rows `#(12 14 16)` and `#(21 42 24)`

whereas

TwoDList columns: 3 rows: 5

returns TwoDList (nil nil nil nil nil nil nil nil nil nil nil nil).

Accessing a TwoDList

Just like Array, TwoDList is accessed by at: and at:put:. The difference is that the at: argument is normally a Point because a position in a table requires two coordinates. The x part of the point is the column number, and the y part is the row number. The following example illustrates the principle:

```
| table |
table := TwoDList on: #(12 14 16 21 42 24) columns: 3 rows: 2.
Transcript clear; cr; show: (table at: 1@2) printString. "Prints 21 – element in column 1, row 2."
table at: 2@2 put: 77. "Changes table to TwoDList (12 14 16 21 77 24) and returns 77."
```

The internal conversion from the Point argument to the position in the underlying array uses the following method:

integerIndexFor: aPoint

$^(\text{columns} * (\text{aPoint y} - 1)) + \text{aPoint x}$

which confirms that elements are stored row-wise because 1@1 is converted to index 1, 2@1 is converted to index 2, and so on.

The at: message can also be used with an integer argument, in which case the table is treated as a one-dimensional array and accessed directly.

Example. Implementing a two-dimensional matrix with TwoDList

As we already mentioned, matrices are very important for computer graphics because they are the basis of transformations of geometric objects such as shifting, scaling, rotation, and mirroring. All these operations can be performed by a combination of matrix scaling, multiplication, addition, and subtraction and implementation of a class capable of these operations would thus be very useful. Creating a framework with a truly sophisticated set of classes supporting matrices and related concepts is beyond the available space and expected mathematical prerequisites and we will thus limit ourselves to designing a simple Matrix class supporting scaling, addition, and transposition.

The first question is where to put Matrix in the class hierarchy. Since Matrix has use for much of TwoDList behavior, we will make Matrix a subclass of TwoDList. Its basic functionality - creation, accessing, and even transposition - are inherited from TwoDList and all we have to do is define methods for scaling and addition.

$$\begin{array}{c} x * \end{array}
 \begin{array}{|c|c|c|} \hline a_{1,1} & a_{2,1} & a_{3,1} \\ \hline a_{1,2} & a_{2,2} & a_{3,2} \\ \hline a_{1,3} & a_{2,3} & a_{3,3} \\ \hline a_{1,4} & a_{2,4} & a_{3,4} \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|} \hline x*a_{1,1} & x*a_{2,1} & x*a_{3,1} \\ \hline x*a_{1,2} & x*a_{2,2} & x*a_{3,2} \\ \hline x*a_{1,3} & x*a_{2,3} & x*a_{3,3} \\ \hline x*a_{1,4} & x*a_{2,4} & x*a_{3,4} \\ \hline \end{array}$$

Figure 7.11. Matrix scaling.

To *scale* a matrix by a number, all its elements are multiplied by the number (Figure 7.11). This can be implemented by enumeration via the collect:, multiplying all elements by the scaling factor:

scaledBy: aNumber

"Create a new matrix whose elements are elements of self multiplied by aNumber."
 $^{\text{self collect: [:el | el * aNumber]}$

Unfortunately, when we test this on an example such as

```
| matrix |  
matrix := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.  
matrix scaledBy: 3
```

we fail miserably, getting a walkback (Exception window) which says that “this class does not support variable size allocation” and indicates that it failed in the new: message which was sent by collect:. The problem is that message new: cannot be used if all parts of the class are not accessed by index. We could have anticipated a problem had we compared the definition of Array

```
ArrayedCollection variableSubclass: #Array  
instanceVariableNames: "  
classVariableNames: "  
poolDictionaries: "  
category: 'Collections-Arrayed'
```

with the definition of Matrix which is

```
TwoDList subclass: #Matrix  
instanceVariableNames: "  
classVariableNames: "  
poolDictionaries: "  
category: 'Book'
```

This is an important lesson because it tells us that some collections don’t understand the collect: message and even the new: message! If collect: does not work with Matrix because it is not just a collection, it should work with the collection component of Matrix. We will thus rewrite our method to create a new scaled collection from the receiver’s collection and then create a new Matrix from it as follows:

```
scaledBy: aNumber  
“Return a new matrix whose elements are elements of self multiplied by aNumber.”  
| newCollection |  
“Modify existing collection.”  
newCollection := collection collect: [:el | el * aNumber].  
“Create new instance with the same size and shape and with the new collection.”  
^self class  
on: newCollection  
columns: self columnSize  
rows: self rowSize
```

This indeed works as it should as you can test by executing the following example:

```
| matrix |  
matrix := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.  
matrix scaledBy: 3
```

As an alternative solution, we will now implement the same operation by enumeration over indices. There are two ways to do this: We can either calculate over the two-dimensional coordinates, or we can access the elements of the underlying array directly. The first approach requires two nested loops (over rows and columns), the second approach depends on the knowledge of the internal representation of TwoDList. The second approach is dangerous because it depends on internal representation and if the implementation of TwoDList changed, our method would stop working. We will thus use the first approach:

```
scaledBy: aNumber  
“Return a new matrix whose elements are elements of the receiver multiplied by aNumber.”  
| scaled |  
“Create a new Matrix object with the appropriate size and shape.”  
scaled := self class columns: self columnSize rows: self rowSize.  
“Calculate its elements from the elements of the receiver.”
```

```
1 to: self rowSize do: [:rowIndex |
  1 to: columnSize do:
    [:columnIndex | |point| point := columnIndex @ rowIndex.
      scaled at: point put: (self at: point) * aNumber]].
^scaled
```

This definition is more complicated than the first one but it works.

To *add* two matrices together, elements from the corresponding cells are added together (Figure 7.12).

a _{1,1}	a _{2,1}	a _{3,1}	+	b _{1,1}	b _{2,1}	b _{3,1}	=	a _{1,1} + b _{1,1}	a _{2,1} + b _{2,1}	a _{3,1} + b _{3,1}
a _{1,2}	a _{2,2}	a _{3,2}		b _{1,2}	b _{2,2}	b _{3,2}		a _{1,2} + b _{1,2}	a _{2,2} + b _{2,2}	a _{3,2} + b _{3,2}

Figure 7.12. Matrix sum. Both matrices must have the same number of rows and columns.

We can thus define matrix addition as follows:

1. Create a new uninitialized matrix with the same size and shape as the receiver.
2. For each point in the range calculate the sum of the corresponding elements of the receiver and the argument and put this value at the corresponding position of the sum.
3. Return the sum.

The implementation is

+ aMatrix

"Calculate a new matrix whose elements are sums of corresponding receiver and argument elements."

```
| sum |
sum := self class columns: self columnSize rows: self rowSize.
1 to: self rowSize do: [:rowIndex | "Enumerate over rows first."
  1 to: self columnSize do: [:columnIndex | | point |
    point := columnIndex @ rowIndex.
      sum at: point put: (self at: point) + (aMatrix at: point)]]].
^sum
```

and execution of the following program fragment with *inspect* confirms that it works:

```
| matrix1 matrix2 |
matrix1 := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.
matrix2 := Matrix on: #(6 3 8 5 9 7) columns: 3 rows: 2.
matrix1 + matrix2
```

Main lessons learned:

- Class TwoDList implements a two-dimensional array with dependency.
- The main purpose of TwoDList is to serve as the value holder for table widgets but its behavior can be extended to implement the mathematical concept of a matrix.
- Some collections don't understand all enumeration messages.

Exercises

1. How can you determine whether a particular collection class has fixed size?
2. Define and test Matrix method multipliedBy: implementing matrix multiplication. Add printing.
3. The concept of a mathematical matrix is different from the intended use of TwoDList. Redefine Matrix independently of TwoDList. Provide the following protocols: creation, accessing, arithmetic (addition, subtraction, negation, scaling, multiplication), and printing.
4. Define a three-dimensional matrix class called ThreeDArray with the protocols listed in Exercise 3.
5. We found that collect: does not work for Matrix. This raises several questions:
 - a. Does do: work?

- b. Which other basic Collection messages don't work with Matrix?
 - c. Which other collection classes suffer from the same problem as Matrix?
 - d. How can we redefine collect: for Matrix?
 - e. What is the advantage of putting TwoDList in the Collection hierarchy if it does not understand collect:?
6. Implement matrix addition using with:do:.
 7. Our matrix addition does not check whether the two matrices can be added. Add such a check.

7.7 Implementing an n-dimensional array

Although most applications use one- or two-dimensional arrays, three-dimensional arrays are also useful (for example in three-dimensional graphics) and higher dimensional arrays have their uses as well. It is thus natural to extend our definitions and define a class that can implement arrays of *any* dimension. We will now implement such a class under the name NDArry in two different ways.

Class NDArry will have the following responsibilities:

- *Creation* - create an uninitialized NDArry with the specified number of dimensions and 'shape' (size of individual dimensions). This will be implemented with message dimensions: anArray where anArray is an array of integers specifying sizes along individual axes. As an example

NDArry dimensions: #(3 4)

will create a two-dimensional array with 3 by 4 elements.

- *Accessing* - get or set an element specified by its indices. The two messages will be atIndexArray: anArray and atIndexArray: anArray put: anObject. As an example, if x is an instance of NDArry,

x atIndexArray: #(3 4)

will return the element with indices 3 and 4.

- *Arithmetic* - negation, addition, subtraction, and multiplication (messages negated, +, -, and *)
- *Printing* (implemented by printOn: aStream). The style of printing will be 'change the first index first'. As an example, a three-dimensional array with dimension sizes #(2 3 4) will be printed as follows:

```
element at index 1 1 1
element at index 2 1 1
element at index 1 2 1
element at index 2 2 1
element at index 1 2 1
element at index 2 2 1
element at index 1 3 1
element at index 2 3 1
element at index 3 3 1
element at index 1 1 2
element at index 2 1 2
etc.
```

- Any auxiliary protocols required to implement the above functionality.

In the following, we will consider two approaches to this problem. One is based on the fact that all collections are internally stored as sequential collections, and the other on the fact that an n-dimensional collection is an array of n-1 dimensional collections.

Solution 1: n-dimensional array as a mapping to one dimension

In this arrangement, the n indices of the n -dimensional array are mapped into a single index of the corresponding one-dimensional array using some mathematical formula. Since the formula is not obvious, we will feel our way through to it by starting with a solution for small dimensions and generalizing the experience to n dimensions.

In the *one-dimensional case*, no mapping is needed – a one-dimensional array is stored as a one-dimensional array and the counterpart of element a_i in the original array is element a_i in the ‘storage’ array. Index i maps into index i .

In the *two-dimensional case*, we already have experience with TwoDList and we know how it is mapped. For a TwoDList with m columns, element $col@row$ is stored in location $(row-1)*m + col$. We conclude, that when we are traversing the storage, ‘the first index changes first’. We will apply this principle to any number of dimensions.

Consider now the *three-dimensional array* in Figure 7.14. To access the shaded element with indices 4,3,1 (column, row, ‘plane’), we must first skip all elements of the first two-dimensional plane in the foreground ($1*(4*3)$ elements), then the first two rows of the background plane ($2*4$ elements), and then access the fourth element in the row. In general, if the number of elements along the column, row, plane dimensions is m , n , and k , the formula to access an element with indices (a, b, c) is $(c-1)*m*n + (b-1)*m + a$ or $a + (b-1)*m + (c-1)*m*n$. It is useful to realize that the first element of the sum corresponds to the first dimension, the second to the second dimension, and the last to the last dimension.

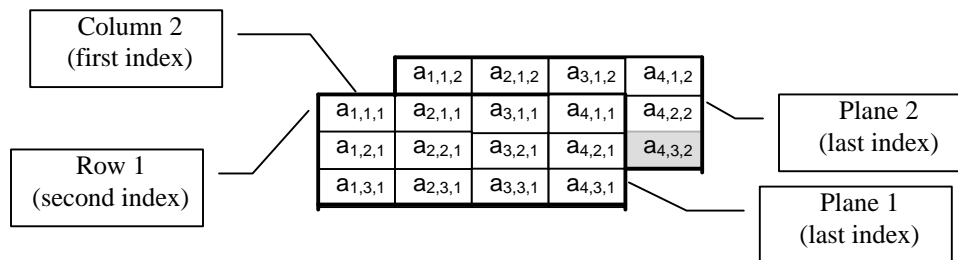


Figure 7.14. A three-dimensional array with dimensions 4, 3, 2 (columns, rows, planes).

We can now deduce that if a an n-dimensional array has dimension sizes $a_1, a_2, a_3, \dots, a_n$, the formula for converting an n-dimensional index $i_1, i_2, i_3, \dots, i_n$, into a one-dimensional index i is

$$i = (i_n - 1) * a_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1 + (i_{n-1} - 1) * a_{n-2} * a_{n-3} * \dots * a_1 + (i_{n-2} - 1) * a_{n-3} * \dots * a_1 + \dots + (i_3 - 1) * a_2 * a_1 + (i_2 - 1) * a_1 + i_1$$

To confirm, at least partially, that this formula is correct, check that it gives the correct result for the cases that we investigated above - one-, two-, and three-dimensional arrays.

After finding the general formula, we will now improve it to speed up the calculation of the index. To do this, we will multiply out the parenthesized expressions, separate the constants, and keep them in instance variables of each instance of NDArry. We find that

$$\begin{aligned} i &= (i_n - 1) * a_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1 + (i_{n-1} - 1) * a_{n-2} * a_{n-3} * \dots * a_1 + (i_{n-2} - 1) * a_{n-3} * \dots * a_1 + \dots + (i_3 - 1) * a_2 * a_1 + (i_2 - 1) * a_1 + i_1 \\ &= (i_n * a_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1 + i_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1 + \dots + i_2 * a_1 + i_1) - (a_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1) - (a_{n-2} * \dots * a_1) - \dots - a_1 \\ &= (i_n * c_n + i_{n-1} * c_{n-1} + i_{n-2} * c_{n-2} + \dots + i_2 * c_2 + i_1 * c_1) - C \end{aligned}$$

where

$$c_n = a_{n-1} * a_{n-2} * a_{n-3} * \dots * a_1$$

$$c_{n-1} = a_{n-2} * a_{n-3} * \dots * a_1$$

etc.

$$c_3 = a_2 * a_1$$

$$c_2 = a_1$$

$$c_1 = 1$$

and

$$C = c_2 + c_3 + \dots + c_n \quad \text{"Does not include } c_1\text{"}$$

After establishing this theoretical background, it remains to identify the internal attributes of NDArry. An NDArry needs to know its elements and we will keep them in instance variables `elements` - an Array of sufficient size. We also need to know the sizes of the individual axes (instance variable `sizes` - an Array of integers), and the constants calculated according to the formulas derived above. These constants will be held in instance variables `multipliers` (an Array of c_i constants) and `constant` (holding the value of C). Finally, we will find it useful to have an instance variable to hold the number of dimensions (`dimensions`). With this, we can now proceed to implementation.

Implementation

Creation protocol. Method dimensions: anArray will return an NArray with properly initialized instance variables but no elements. We will define it in the usual way:

dimensions: anArrayOfSizes

^self new dimensions: anArrayOfSizes

where instance method dimensions: performs all initialization as follows:

dimensions: anArrayOfSizes

```
"Initialize, including calculation of internal constants from specification."  
| temp |  
dimensions := anArrayOfSizes size.  
sizes := anArrayOfSizes.  
"The total number of elements is equal to the product of all dimensions."  
elements := Array new: (anArrayOfSizes inject: 1 into: [:size :el | el * size]).  
multipliers := Array new: dimensions.  
multipliers at: 1 put: (temp := 1).  
2 to: dimensions  
do:  
    [:index |  
        temp := temp * (anArrayOfSizes at: index - 1).  
        multipliers at: index put: temp.].  
constant := multipliers inject: -1 into: [:sum :el | el + sum]
```

Test that the method works correctly.

Accessing protocol. To identify an element, we must specify all its indices. The argument of both the *get* and the *put* method must therefore be an n-dimensional array of indices. If we called our methods *at:* and *at:put:*, this could cause some problems because conventional *at:* and *at:put:* methods expect an integer argument. To avoid this, we will thus use names *atIndices:* and *atIndices:put:*. The definition of *atIndices:* is a simple implementation of the formulas derived above:

atIndices: anArray

```
"Return element specified by its n indices."  
| index |  
index := 0.  
multipliers with: anArray do: [:el1 :el2 | index := index + (el1 * el2)].  
^elements at: index - constant
```

We took advantage of the *with:do:* enumeration method which operates on two collections simultaneously. Method *atIndices:put:* is a simple extension of this definition. Since the calculation of the index is the same in both accessing method, it would be advantageous to define it as a special method and share it. When you are finished, create an NArray and test its accessing protocol.

Arithmetic protocol. Given our accessing protocol, arithmetic is easy and we will leave it as an exercise.

Printing. Assume that we want to print the contents of a NArray with dimensions #(3 4 6) as

```
1 1 1: e "Element at index 1 1 1"  
2 1 1: e "Element at index 2 1 1"  
3 2 1: e "Element at index 3 2 1"  
1 3 1: e "Element at index 1 3 1"  
2 3 1: e "Element at index 2 3 1"  
3 3 1: e "Element at index 3 3 1"  
1 1 2: e "Element at index 1 1 2"  
2 1 2: e "Element at index 2 1 2"  
3 1 2: e "Element at index 3 1 2"
```

1 2 2: e "Element at index 1 2 2"
etc. (comments added only for explanation)

in other words, we again use the principle 'first index changes first'.

To obtain the proper behavior of `printString`, we must redefine the `printOn:` method. In our example, the desired output consists of a pattern of indices followed by the element at that index. We will implement this by constructing and continuously updating the index pattern, printing it, printing the element, and updating a pointer into the elements array. Pattern updating will be implemented with a method that returns `nil` when the last pattern has been printed:

printOn: aStream

```
"Append to the argument, aStream, the elements of the Array enclosed by parentheses."  
| pattern position |  
position := 1.  
aStream nextPutAll: 'NDArray'; nextPutAll: ' ('; cr.  
pattern := (Array new: dimensions) atAllPut: 1.  
[pattern isNil] whileFalse:  
    [1 to: pattern size do: [:index | aStream nextPutAll: (pattern at: index) printString, ' '].  
    aStream nextPutAll: ': ', (self atIndices: pattern) printString; cr.  
    position := position + 1.  
    pattern := self nextPattern: pattern].  
aStream nextPut: $)
```

The process of updating the pattern is as follows: Start from the left end of the current pattern and go right, looking for a number whose value is smaller than the value of the corresponding dimension. If such an element exists, increment it by 1 and reset all elements to the left to 1; return the resulting new pattern. If such an element does not exist, return `nil` to indicate the end of the process. This algorithm is implemented by the following private method:

nextPattern: anArray

```
"Find the next pattern of indices, return nil if there is none."  
| nextPattern |  
nextPattern := anArray copy.  
"Look for the first element that is smaller than the corresponding dimension."  
1 to: dimensions do:  
    [:index | | value |  
        value := anArray at: index.  
        value < (sizes at: index)  
            ifTrue: [ "Increment the element, reset the preceding, return result."  
                nextPattern at: index put: value + 1.  
                nextPattern atAll: (1 to: index - 1) put: 1.  
                ^nextPattern]].  
"All patterns printed, we are finished."  
^nil
```

Solution 2: n-dimensional array as an array of n-1 dimensional arrays

A completely different view of an n-dimensional array is as a one-dimensional array of n-1 dimensional arrays (Figure 7.14). This is a recursive view in which a structure contains objects of similar structure but simpler and the bottom object in the hierarchy is a one-dimensional array which contains the actual elements. Although this idea appears complicated, its implementation is simpler than Solution 1. To distinguish our two approaches, we will call this class `NDArrayRecursive`.

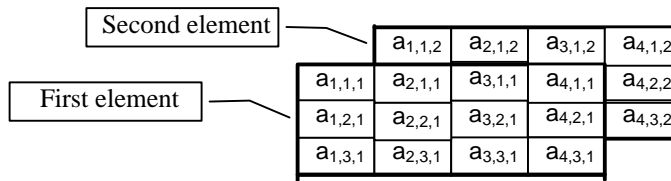


Figure 7.15. Three-dimensional array (three indices) as an array of two-dimensional arrays.

The class will have similar protocol as in Solution 1 and its instance variables will also be similar. They will include `dimensions` (integer number of dimensions), `sizes` (array of sizes of individual dimensions), `elements` (array of n-1 dimensional arrays for n > 1, array of elements for n = 1). Our printing strategy will again be based on a pattern array holding indices and this array will be held in instance variable `pattern`. Since several of the methods will be recursive, it will be useful to keep track of the number of dimensions of the whole matrix and we will use instance variable `globalSize` for this purpose.

Implementatio of protocols

Creation. We need two creation messages: One to create the topmost matrix, and another to create the matrices inside it. The reason for this is that we want all enclosed matrices know the size of the outermost matrix.

The topmost creation method will be called `newWithDimensions: anArray` where `anArray` is the array of dimensions as in Solution 1. This method recursively creates lower level matrices by message `dimensions: anArray globalSize: anInteger`. The definitions of the top level class creation method is

newWithDimensions: anArray

"Return new instance with axe dimensions as specified"

^self new dimensions: anArray globalSize: anArray size

and the corresponding class method for creating the enclosed lower level matrices is

dimensions: anArray globalSize: anInteger

"Return new instance with axe dimensions as specified"

^self new dimensions: anArray globalSize: anInteger

This method sends the instance message `dimensions: globalSize:` which initializes the created object:

dimensions: anArray globalSize: anInteger

"Initialize elements from specification."

| size |

sizes := anArray.

dimensions := anArray size.

globalSize := anInteger.

size := anArray at: 1.

elements := Array new: size.

dimensions > 1 ifTrue: "Create the lower level matrix."

[1 to: size do:

[:index | elements at: index put: (NDArrRecursive

dimensions: (anArray copyFrom: 2 to: dimensions)

globalSize: anInteger)]]

The block statement creates the appropriate number of lower dimensionality arrays if dimension is greater than 1 and passes the remaining dimensions down. As an example, if we want to create a three-dimensional array with sizes `#(4 6 2)`, the creation method creates an `NDArrRecursive` with dimension = 3, size = 4, and three elements, each of them a two-dimensional `NDArrRecursive`. Each of these arrays has dimension 2 (it is two-dimensional), size = 6, and contains six one-dimensional arrays. Each of these one-dimensional arrays has two uninitialized elements. All share the same `globalSize` = 3.

When we tried to test the creation mechanism with `inspect`, we got an exception because inspecting elements requires the `do:` message which class `Collection` leaves as subclass responsibility. We thus added `do:` as follows:

do: aBlock

"Evaluate aBlock with each of the receiver's elements as the argument."

```
dimensions = 1  ifTrue: [elements do: [:element | aBlock value: element]]
                ifFalse: [elements do: [:element | element do: aBlock]]
```

We then inspected

```
NDArrayRecursive dimensions: #(2 3 4)
```

and the result was as expected.

Accessing. Here again, we must distinguish higher dimension arrays whose elements are lower dimensionality arrays, and the actual data stored in the `NDArray`:

atIndices: anArray

```
^dimensions = 1  ifTrue: [elements at: (anArray at: 1)]
                ifFalse: [(elements at: (anArray at: 1))
                           atIndices: (anArray copyFrom: 2 to: dimensions)]
```

Note that we must use message `atIndices:` on the last line - message `at:` would fail because the argument is an array rather than an integer. Note also that we remove the first index from the elements array as we pass it to the `n-1` dimension array. Method `atIndices:put:` is similar and we leave it as an exercise. We tested accessing by executing the following test program with *inspect*

```
| nda |
nda := NDArrayRecursive dimensions: #(2 3 4).
1 to: 2 do:
    [:col| 1 to: 3 do:
        [:row| 1 to: 4 do:
            [:plane| nda atIndices: (Array with: col with: row with: plane)
                           put: col*row*plane]]].
```

nda

and everything works.

Printing. For this implementation, we will write code to output the following format (produced for the array from the test code above):

```
NDArrayRecursive (
1 1 1 : 1
1 1 2 : 2
1 1 3 : 3
1 2 1 : 2
1 2 2 : 4
1 2 3 : 6
1 3 1 : 3
1 3 2 : 6
1 3 3 : 9
2 1 1 : 2
2 1 2 : 4
2 1 3 : 6
2 2 1 : 4
2 2 2 : 8
2 2 3 : 12
2 3 1 : 6
```

```
2 3 2 : 12
2 3 3 : 18
)
```

Our solution will again be based on an pattern of indices but we will leave all printing to the lowermost NDArry level - the one-dimensional array. However, all levels in the hierarchy must contribute to the pattern as they increment their indices and we will implement this as follows:

- When a request to execute `printOn:` arrives to the uppermost n-dimensional NDArry, this object initializes `pattern` to an array consisting of all 1s. It then passes `pattern` down to its n-1 dimensional elements which pass it down to their components, and so on. The one-dimensional at the bottom NDArry prints the indices and the values.
- During each iteration through the printing loop of a k-dimensional NDArry, the NDArry object updates `pattern` by incrementing its k-th component, and resetting the elements to the right to 1. This is done recursively and all indices are thus correctly updated.

The implementation of this strategy is as follows:

printOn: aStream

"Append to the argument, aStream, the elements of the Array enclosed by parentheses."

"The top level array prints the name of the class, etc."

`dimensions = globalSize`

`ifTrue: [aStream nextPutAll: 'NDArryRecursive'; nextPutAll: ' ('; cr.`

`"Initialize index pattern and pass it down to lower level arrays."`

`pattern := (Array new: dimensions) atAllPut: 1.`

`elements do: [:element | element pattern: pattern]].`

`dimensions > 1`

`ifTrue: [elements do: [:element |`

`aStream nextPutAll: element printString.`

`self updatePattern]]`

`ifFalse: [pattern size timesRepeat:`

`[1 to: pattern size do: [:index | aStream nextPutAll: (pattern at: index) printString , ' '].`

`aStream nextPutAll: ': ', (elements at: pattern last) printString; cr.`

`self updatePattern]].`

"Top level array prints the closing bracket."

`dimensions = globalSize ifTrue: [aStream nextPut: $)]`

where

pattern: anArray

"Initialize this pattern and pass it down to lower level patterns."

`pattern := anArray.`

`dimensions > 1 ifTrue: [elements do: [:element | element pattern: anArray]]`

Finally, method `updatePattern` simply assigns a new pattern to `pattern`, and updating is done as follows:

updatePattern

"Increment my pattern index by 1 and reset those to the right to 1."

`| myIndex |`

`myIndex := globalSize - dimensions + 1.`

`pattern at: myIndex put: (pattern at: myIndex) + 1.`

`pattern atAll: (myIndex + 1 to: globalSize) put: 1`

This completes the implementation. We conclude that the solution based on recursion is simpler – we did not even have to figure out how to calculate the index from the index array. It would be interesting to compare the speed and memory requirements of the two solutions.

Main lessons learned:

- Recursion may be applied to structure, not only to algorithms.
- Recursive implementations of algorithms and objects are often simpler.

Exercises

1. Compare the number of calculation required to calculate the index before and after the simplification of the formula.
2. Implement the missing protocols in each implementation of NDAarray.
3. Implement printing of the first solution by accessing elements directly. Compare speed.
4. Re-implement printing for the second solution so that the first index changes first.
5. Compare the speed of element accessing of both implementations.

7.8 Use of TwoDList in the Table widget

To conclude the chapter, we will now present the table widget, both as an example of the use of sequenceable collections and as a useful GUI component.

The table is a read only widget (you cannot change the contents of its cells) for displaying objects in two-dimensional form. A closely related widget is a data set which also displays data in the form of a table but allows the user to modify cell values. Also, while a table can display any mixture of objects, a data set is used to display rows of related multi-item objects such as student records, or book information. In both the table and the data set, the user can select a cell, a row, or a column, and the program can monitor selection changes and respond appropriately.

The *Aspect* of a table is a TableInterface object (Figure 7.16) with

- information about the data displayed in the table, and the current selection (combined into a SelectionInTable object),
- a description of the table's row and column labels and its other visual aspects.

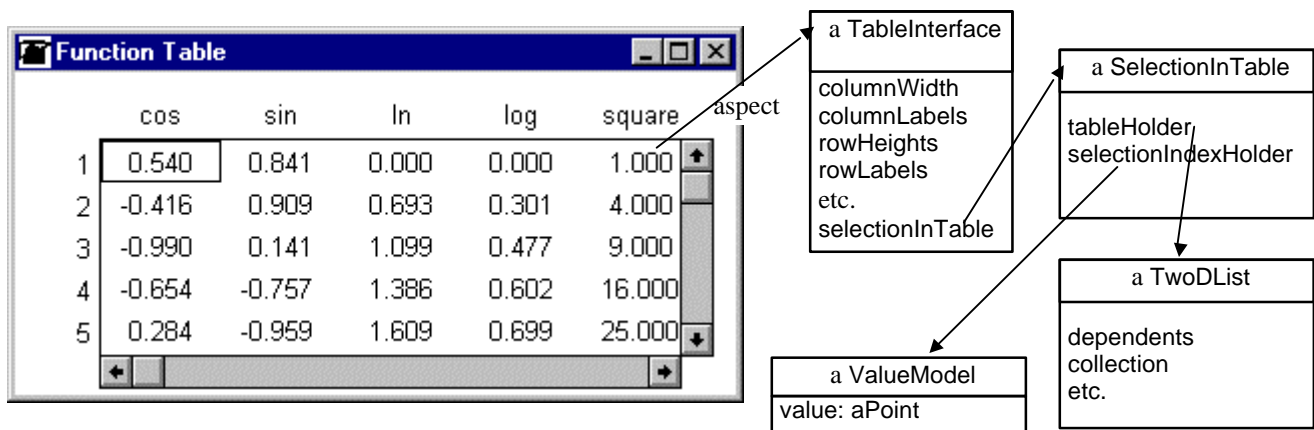


Figure 7.16. The structure of table widget support.

The following slightly edited comment of TableInterface captures its main properties.

"A TableInterface is used to hold the information needed for controlling the format and content of a TableView. The tabular structure of the underlying data is usually captured in terms of a SelectionInTable. TableInterface is also a point at which many operations may be done to the table component. Any operations that are needed to be done to either the row labels or column labels of a table should go through the TableInterface.

Operations that have to deal with the visual properties of a table (color and emphasis) on a column, row, or element basis should either be done in the same way that the TableInterface does it, or through the TableInterface.

Instance Variables:

columnWidths	<ArrayedCollection of <Integer> Integer nil> The widths of the columns
columnFormats	<Array Symbol nil> #left, #right, and #centered are format symbols
columnLabels	<ValueHolder with: <TableAdaptor> nil> Labels for the columns of the Table
columnLabelsFormats	<Array Symbol nil> #left, #right, and #centered are format symbols
rowLabelsWidth	<Integer nil> The width of the row label display
rowLabels	<ValueHolder with: TableAdaptor nil> Labels for the rows of the Table
rowLabelsFormat	<Symbol nil> #left, #right, or #centered
selectionInTable	<SelectionInTable> The underlying data in tabular form
columnLabelsAndSelection	<SelectionInTable> Models the selection of an entire column
rowLabelsAndSelection	<SelectionInTable> Models the selection of an entire row

A TableInterface allows you to control the width of columns in pixels, and the format (left justified, right justified, or centered) of the contents of the cells and the row and column labels. It also allows the user to select either an individual cell or a whole row or column.

Example: Table of functions

Problem: Write an application displaying a table of values of sin, cos, ln, log, and square for all values from 1 to 10 in steps of 1. The desired format is shown in Figure 7.16.

Solution: To solve this problem, we must paint the user interface, define its parameters, and define the required methods.

Painting the table and defining its parameters. This step is routine. Draw the outline of the table, enter the name of the *Aspect* (name of method returning a TableInterface - we called it tableInterface) and specify any *Details* you want to override the defaults. *Define* the *Aspect* of the table. The labels and the number of columns and cells will be assigned by the initialize method.

Defining the methods. Since our application is so simple, we only need an initialize method to create the all-important TableInterface object from its pieces: A TwoDList with appropriate size and values, and the TableInterface object itself with a SelectionInTable containing the TwoDList object as its underlying collection value holder. Finally, we will specify visual aspects of the table including column widths, text format, and row and label column:

initialize

"Create objects underlying the Table widget and calculate its parameters."

| functions values columnWidths |

"Create a TwoDList to hold the data and provide dependency based binding to the widget."

values := TwoDList columns: 5 rows: 10.

"Calculate the data to be displayed, format it, and insert it into the TwoDList."

functions := #(#cos #sin #ln #log #squared).

1 to: values columnSize do:

[:column | 1 to: values rowSize do:

[:row | values atPoint: column @ row

put: (PrintConverter print: (row perform: (functions at: column))

formattedBy: '####.###')]].

"Create the TableInterface and associate it with the list of values."

tableInterface := TableInterface new selectionInTable: (SelectionInTable with: values).

"Define labels and column widths of the TableInterface."

columnWidths := 60. "All columns have the same width."

tableInterface columnLabelsArray: #('cos' 'sin' 'ln' 'log' 'square');

rowLabelsArray: #(1 2 3 4 5 6 7 8 9 10);

columnWidths: columnWidths

This definition requires several comments:

1. Since we access the TwoDList via rows and columns, we use accessing method `atPoint:` `aPoint` put: `anObject`. Be careful to specify `aPoint` correctly as `column@row`. In our first attempt, we accidentally reversed the order of row and column and obtained a strange result.
2. To calculate cell values, we stored the names of the necessary methods as Symbols in an array called `functions`. We then sent the messages by using the column number to extract the appropriate function from the array, and sending `perform:` with the value of the row as the argument.
3. After calculating a cell value, we used the `PrintConverter` to convert the number to a suitable format - the default format is not satisfactory.
4. The default display of the table is awkward and does not have any labels. To obtain the desired look, we ask the `TableInterface` to set suitable column widths, and define appropriate labels.

Main lessons learned:

- A table is a read-only widget.
- The *Aspect* of a table is an instance of `TableInterface`.
- `TableInterface` has two functions: It holds the displayed data and the current selection, and contains information about the visual aspects of the table such as column width and row and column labels.

Conclusion

Most computer applications work with collections of objects. Collections can be divided according to several classification criteria and VisualWorks library contains many collection classes, some for system needs, others for general use.

Abstract class `Collection` at the top of the collection hierarchy defines several protocols shared by all or most collections. Besides creation, the most important of these is probably enumeration - iteration over all elements of the collection. Enumeration messages are used in most iterations over collections and experienced programmers use their specialized forms whenever opportunity arises.

The collections most useful for general use are array, ordered and sorted collection, list, string, symbol, set, bag, and dictionary. The main differences between these collections are whether they are restricted to a fixed size, whether they allow duplication, and whether their elements are ordered and how this ordering works. The class hierarchy uses ordering as the distinguishing feature to divide collections into two main groups - sequenceable and non-sequenceable.

The main protocols shared by all collections include creation, enumeration, accessing, testing, adding and removing elements (if allowed), and conversion to another kind of collection. Conversion is often used to perform an operation such as sorting or elimination of duplication.

Beginners usually encounter problems when using certain collection messages. For proper use, make sure to know which object the message returns (some methods return the argument when one might expect the receiver), and whether the method changes the receiver or returns its modified copy while leaving the receiver unchanged.

In this chapter, we started our presentation of collections with class `Array`. An array is a fixed-size sequenceable collection which means that its size cannot grow or shrink, and that its elements are accessed by consecutive integer indices numbered from 1. Arrays are most commonly created as literals, by conversion from other types of collections, by `new:` followed by calculation of individual elements, or by one of the `with:` messages. When an array is created by the `new:` creation message, all its slots are filled with nil objects. Arrays - like most other Smalltalk collections - allow their elements to be any objects or even a mixture of different kinds of objects. The main advantages of arrays over other sequenceable collections are their ease of creation, compactness in terms of internal representation, and fast access.

Each array access in Smalltalk checks whether the specified index is within the array's bounds. This checking adds some execution overhead but makes use of arrays safe.

Arrays are not used as much in Smalltalk as in other languages because Smalltalk provides many other types of collections which may be more suitable for the task at hand, and because it is often preferable to define a class with named components to hold the values instead of using collections. The test that determines whether a multi-component object should be represented by a collection or a class with named

variables is whether the components have identifiable, 'name-able' roles or whether they are just a collection of objects of the same kind.

A `TwoDList` is a collection designed to support table widgets. It can be considered an extension of `Array` in two ways: Its elements are accessed by a pair of indices representing a row and a column number, and it implements dependency. We showed how `TwoDList` could be used to create a new class implementing mathematical matrices with minimal effort.

The table widget is a read-only two-dimensional GUI component equipped with vertical and horizontal scroll bars and visual features including row and column labels, programmable cell width and height, and formatting of labels and cell contents. Its underlying *Aspect* object is a `TableInterface`. A `TableInterface` holds information related to the appearance of the table such as labels and column widths, and a `SelectionInTable` object containing the displayed data and the current selection. The data itself is represented by a `TwoDList`.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Array, *Association*, *Bag*, **Collection**, **Dictionary**, **List**, **OrderedCollection**, *SelectionInTable*, **Set**, **SortedCollection**, **String**, **Symbol**, **TableInterface**, **Text**, **TwoDList**.

Widgets introduced in this chapter

Table.

Terms introduced in this chapter

array - fixed-size collection whose elements are indexed by consecutive integers starting at 1

bag - unordered collection that keeps track of the number of occurrences of its elements

bounds checking - testing whether an index of a sequenceable collection falls within acceptable limits

collection - group of nameless elements with creation, accessing, testing, adding, deletion, and enumeration protocols

dictionary - collection of key-value pairs

enumeration - iteration over collection elements

index - integer number used to access elements of sequenceable collections

nested collection - collection used as an element of another collection

ordered collection - collection whose elements are stored in a fixed externally accessible order; in a more restrictive sense, instance of class `OrderedCollection`

set - unordered collection that eliminates duplication of elements

sort block - block used by sorted collection to determine the order of its elements

sorted collection - a sequenceable collection whose elements are ordered according to a sort block

sequenceable collection - a collection whose elements are arranged in a fixed order defined by integer indices

string - a sequence of character codes with no information about their rendition

table - a fixed size read-only two-dimensional widget for displaying heterogeneous objects in tabular form

text - string with emphasis prescribing how the string should display itself

unordered collection - collection whose internal ordering is determined by the implementation and inaccessible to the user

Chapter 8 - More Sequenceable Collections, List widgets

Overview

In this chapter, we conclude our exploration of sequenceable collections with `OrderedCollection`, `SortedCollection`, `List`, `String`, `Symbol`, and `Text`. We then present single- and multiple-selection lists. Unordered collections are the subject of the next chapter, and additional widgets including `Dataset`, `Subcanvas`, `Notebook`, `Dialog Window`, and `Menus` are covered in Appendix 2.

8.1 Class `OrderedCollection`

Class `OrderedCollection` is a very frequently used subclass of `SequenceableCollection`. Its instances have the following properties:

- Elements are indexed starting with 1 (as in all sequenceable collection).
- Size (number of elements in collection) may be smaller than capacity (number of slots).
- Capacity automatically grows when size = capacity and a new element is added.
- Elements can be removed, changing size but not capacity.
- Elements are usually accessed by enumeration, at the start, or the end rather than by index.

Ordered collections are used mainly to gather elements whose number is not known beforehand or whose number changes. One example of both situations is a program that collects labels for a list widget when the number of labels is initially unknown. Two common situations of the second kind are stacks and queues. Quoting from the comment of `OrderedCollection` ‘a stack is a sequential list for which all additions and deletions are made at one end of the list; a queue is a sequential list for which all additions are made at one end, but all deletions are made from the other end’. You will immediately recognize that a pile of stacked trays in a cafeteria is an example of a stack, and that a line of customers in a bank is an example of a queue. Stacks and queues are very important in computer applications and we will deal with them in more detail in Chapter 11.

After this general introduction, we will now briefly examine `OrderedCollection` protocols and then give several examples of their use.

Creation

Ordered collections can be created in the same ways as arrays except that there is no concept of a literal `OrderedCollection`. In practice, ordered collections are most often created with `new` (default capacity 5) because the eventual maximum size of the collection is usually unknown. When the approximate size is known beforehand, one should always use `new: to` eliminate time-consuming growing. When a new `OrderedCollection` is created, it is allocated a number of slots equal to the requested capacity but its size is 0 and its response to `isEmpty` is true.

Ordered collections are also often created by the conversion message `asOrderedCollection`, and the `withAll: aCollection` message is occasionally used with the same effect.

Accessing

Although elements of `OrderedCollection` may be accessed by `at:` and `at:put:`, they are mostly accessed by enumeration, at the start, at the end, or by locating the desired element by its value. Some of the messages used for accessing are

<code>first</code>	“Returns the first element without changing the collection.”
<code>last</code>	“Returns the last element without changing the collection.”
<code>addFirst: anObject</code>	“Makes anObject the first element, shifts rest to the right, grows if necessary.”

addLast: anObject	"Adds anObject at end, grows if necessary."
add: anObject	"Same as addLast: anObject."
add: newObject after: oldObject	"Inserts newObject after the first occurrence of oldObject."
removeFirst	"Removes the first element."
removeLast	"Removes the last element."
remove: anObject	"Removes the first occurrence of anObject."
removeAllSuchThat: aBlock	"Removes all elements that satisfy the block."

Note that adding an element at the beginning of an `OrderedCollection` or inside it changes the index of all the remaining elements. As the collection evolves, the index of an element may thus change and this is one of the reasons why `at:` and `at:put:` messages are not much used with ordered collections.

To understand ordered collections and how they differ from arrays, it is useful to know how they are implemented. When an element is removed, it is replaced with `nil` and the `OrderedCollection` adjusts its instance variables `firstIndex` and `lastIndex` which it uses to keep track of its first and last *valid* elements (Figure 8.1). This means that the concept of a position and an index are not equivalent and to deal with this, `OrderedCollection` redefines accessing messages `at:` and `at:put:` on the basis of `firstIndex` and `lastIndex`.

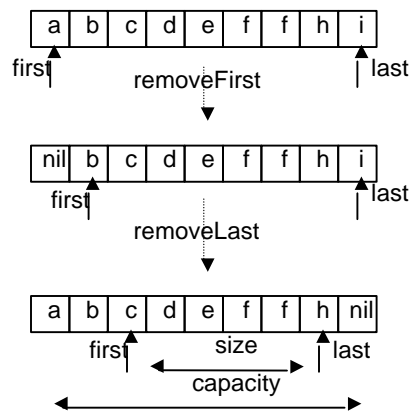


Figure 8.1. Operation of `removeFirst` and `removeLast`.

Here is the definition of `removeFirst`:

removeFirst

"Remove the first element of the receiver. If the receiver is empty, provide an error notification."

```
| firstObject |
self emptyCheck.
firstObject := self basicAt: firstIndex.
self basicAt: firstIndex put: nil.
firstIndex := firstIndex + 1.
^firstObject
```

Message `emptyCheck` first tests whether the collection is empty. If it is, `emptyCheck` causes a *non-proceedable error*, opening an Exception notifier with the *Proceed* button disabled. Clicking *Terminate* exits the execution of the method and avoids getting into further trouble by attempting to access an object at a non-existent index. The *Debug* button remains enabled and you can proceed in the Debugger - only to run into other error conditions.

Note that `removeFirst` does not use the `at:` message, but rather `basicAt:`. The reason for this is that `OrderedCollection` defines its own `at:` and this definition is inappropriate in this context. What we want to use is the definition that works on the basis of the 'regular' index rather than `firstIndex` and `basicAt:` allows us to do that. Method `basicAt:` is defined in `Object` and is equivalent to the usual `at:` message.

To see why we need methods such as `basicAt:`, consider the situation in Figure 8.2 and assume that class `C` needs the top level definition of `at:`. If we used `super at:` we would access the definition in class `A`, presumably different from that in class `Object`. The only way to deal with this situation is to define a *synonym* of `at:` in `Object` and make a gentlemen's agreement with all Smalltalk programmers that they will

never redefine it in any subclass. Method `basicAt:` has exactly this role. If everybody plays by the rules, `basicAt:` thus has a guaranteed single definition that we can always rely on. When you examine the library, you will find that there are quite a few `basic...` methods.

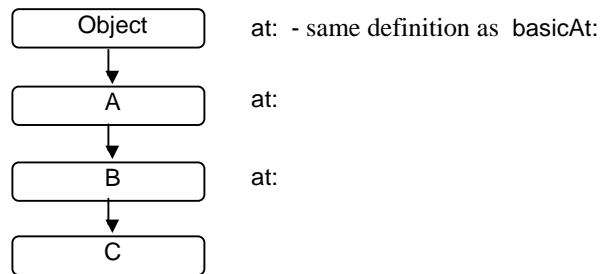


Figure 8.2. The need for `basicAt:`. Class A redefines `at:` but class C needs `at:` from Object.

How much are ordered collections used and how?

To find how much Smalltalk relies on ordered collections, we checked how many methods in the library use them and we found several hundred references to class `OrderedCollection`. This shows that `OrderedCollection` is one of the most heavily used classes. Browsing shows that ordered collections are usually created with `new` and used mainly to gather a collection of elements whose number is initially unknown. The resulting collection is often processed in some way, for example sorted, converted to an array for more efficient processing, or to a set to eliminate multiple copies of duplicated elements. We also checked how many instances of `OrderedCollection` our running session has by executing

`OrderedCollection allInstances size`

We found over 200 references, a large number that will fluctuate greatly as you run an application¹. These findings confirm that ordered collections are very popular and that they are a very important part of the Smalltalk environment.

Main lessons learned:

- Ordered collections are indexed but the index is rarely explicitly used for access. Instead, elements are usually added and retrieved at the end or at the start, preserving the order in which elements are added.
- The size of an ordered collection may be smaller than its capacity.
- Adding a new element to an ordered collection automatically grows the capacity if necessary.
- The growth of an ordered collection is relatively time-consuming and the only way to avoid it is to create the ordered collection with capacity equal to the maximum anticipated need or greater.
- Making an ordered collection larger than necessary wastes a negligible amount of memory but making it too small may cost considerable execution time.
- Removing an element from the beginning of an ordered collection changes the index of remaining elements. It does not shrink the capacity.
- A new ordered collection contains slots but no elements.
- Ordered collections are used mainly to collect elements when their exact final number is unknown and as stacks or queues.

Exercises

¹ Method `allInstances` does not give an accurate picture of all instances of a class because it counts even unreferenced objects that have not yet been garbage-collected.

1. When a new ordered collection is created, it contains slots but no elements. What are these 'slots'? Is something stored in them? (Hint: Check the implementation of at: in OrderedCollection and see if you can examine the unoccupied slots.)
2. Find how to obtain the number of references to a given class. (Hint: The browser knows how to find all references and returns them as a collection.)
3. List at least two useful methods inherited by OrderedCollection from SequenceableCollection and give examples of their use.
4. OrderedCollection and other variable-size collections automatically grow but don't shrink. Define a method to shrink an ordered collection. Comment on the usefulness of such a method.
5. Why is add: in OrderedCollection equivalent to addLast:? (Hint: Which of addFirst and addLast is faster? Why?)
6. Find two other basic... methods and explain their purpose.
7. Write a code fragment to create an OrderedCollection with capacity 5 and three elements. Add four new elements, then delete one at the beginning and one at the end. Print all slots of the collection before and after each step. (Hint: Use basicAt:.)
8. Since removeFirst moves the firstIndex pointer, the 'removed' element cannot be accessed so why bother changing it to nil?
9. Are the slots vacated at the beginning of an OrderedCollection by removeFirst ever reclaimed? If so, when and how?

8.2 Several examples with ordered collections

In this section, we present several examples using ordered collections. They include code from the library, a brief code fragment, and a small application using two classes.

Example 1: Selecting book entries from a library catalog

Problem: Assume that we have a library catalog application with information about books. All book information is stored in a collection accessible via variable books. The collection is an OrderedCollection because it must be able to grow and shrink, and its elements are instances of class Book. Class Book has instance variables title, author, publisher, year, and number (Figure 8.3) and accessing methods for accessing them. A typical task for the application is to extract all books published by 'Addison-Wesley' in 1997.

	title	author	publisher	year	number
book 1	'The personal computer ...'	'Sargent ...'	'Addison-Wesley'	1995	11
book 2	'Principles of computer science'	'Schaffer'	'Prentice-Hall'	1988	22
book 3	'Programmer's guide to the IBM PC'	'Norton'	'Microsoft Press'	1985	13
book 4	'Programming languages'	'Pratt'	'Prentice-Hall'	1965	54
book 5	'Data structures ...'	'Kruse'	'Prentice-Hall'	1994	52
book 6	'Introduction to AI'	'Charniak ...'	'Addison-Wesley'	1986	36
book 7	etc.	etc.	etc.	etc.	etc.

Figure 8.3. Structure of ordered collection books.

Solution: To solve this problem, the code must check the publisher and year components of each instance of Book and select all books that satisfy the given condition. This calculation can be performed very easily with select: as follows:

```
selectedBooks := books select: [:book] (book publisher = 'Prentice Hall') and: [book year = 1997]]
```

Example 2: Conversion to OrderedCollection - how does it work?

Class Collection contains many conversion messages and one of them - asOrderedCollection - converts its receiver - an arbitrary collection - to an OrderedCollection. Here is its definition :

asOrderedCollection

```
| anOrderedCollection |  
anOrderedCollection := OrderedCollection new: self size.  
self do: [:each | anOrderedCollection addLast: each].  
^anOrderedCollection
```

The method first creates a new ordered collection whose capacity is equal to the size of the receiver collection. It then adds all elements of the receiver at the end of the new collection and returns the new collection. The original collection is not affected but shares all its elements with the returned ordered collection.

Example 3: A simple shopping minder application

Problem: This problem is a test of an idea that will be explored in more detail in the next chapter. In the present version, our task is to write a program allowing the user to enter a list of store items and prices and display it in the Transcript one item per line followed by the total price. Items are sorted alphabetically by name and all items that cost more than \$10 are listed behind the total. An example of the desired output is

```
cat food price: 24 dollars 54 cents  
flour price: 4 dollars 75 cents  
sugar price: 3 dollars 15 cents
```

Total price: 32 dollars, 44 cents

Expensive items:

```
cat food price: 24 dollars 54 cents
```

Solution: We will define two classes, one to hold item information, and the other the future application model. Since the problem is quite simple, we can write class specifications without constructing scenarios and performing other preliminary work:

ShoppingMinder: Holds a collection of items, knows how to obtain item information from user and how to display summary information in the Transcript.

Superclass: Object. In the final implementation, Shopping minder will be a subclass of ApplicationModel.

Components:

- items - OrderedCollection of Item objects – the number of items is unpredictable

Contracts and behaviors:

Collaborators

- Creation
 - Create and initialize items to a reasonably large OrderedCollection (capacity 20)
- Accessing
 - Get item information from user via Dialog and create an Item Item
- Adding
 - Add new item to items
- Arithmetic
 - Calculate totals
- Printing
 - Display items Item
 - Display expensive items Item
 - Display total
- Execution
 - Execute shopping process including entry of items and output of results

Item: Holds name and price of item.

Superclass: Object

Components:

- name - name of item (a String)

- price - dollars and cents (a Currency)

Contracts and behaviors:

Collaborators

- Creation
 - Create instance with given name and price
- Accessing
 - Get and set methods
- Printing
 - Return name and price as a suitably formatted String

On the basis of this plan, it is now easy to write the implementation. We will leave most of the work to you and confine ourselves to a few elements.

Class ShoppingMinder

We will start at the execution level. To run the application, the user will execute

ShoppingMinder new execute

This method creates a new instance of ShoppingMinder and initializes its items instance variable. Its definition is

```
new
    ^super new initialize
```

and the initialize method is

```
initialize
    "Initialize items to a suitably sized ordered collection."
    items := OrderedCollection new: 20.
```

Method execute obtains items from the user and displays the results. Its definition is simply

```
execute
    "Get items from the user and display the results."
    [self addItem] whileTrue: [self addItem].
    self displayItems.
    self displayTotal.
    self displayExpensiveItems
```

Note that the formulation implies that addItem returns true when an item is added, and false when the user indicates that there are no more items. Its definition is as follows:

```
addItem
    "Obtain an Item object from the user or an indication that no more items are needed. Add it to items."
    | name price |
    name := Dialog request: 'Enter item name' initialAnswer: ''.
    name isEmpty ifTrue: [^false].          "Exit and terminate loop - user indicated end of entry."
    price := Dialog request: 'Enter item price' initialAnswer: ''.
    price isEmpty ifTrue: [^false].         "Exit and terminate loop - user indicated end of entry."
    items add: (Item name: name price: price asNumber).
    ^true
```

where name:price: is the creation method for Item, to be defined later.

```
displayTotal
    "Add prices of all items and display the total in Transcript."
    Transcript cr; show: 'Total price: '; tab; show:
```

```
(items inject: (Currency cents: 0)
 into: [:total :item | total + item price]) displayString
```

Note the use of inject:into: with Currency objects, and the displayString message - you may have expected printString. displayString is one of the two standard methods that convert an object to a String, the other being the familiar printString. As you know, printString is used mainly to display information during debugging and in the inspector ; in other words, it is intended primarily for the *code developer*. Method displayString, on the other hand, is intended for the *user interface*, for example, when displaying an object in a list widget. Its essence is the same as that of printString (it creates a String or Text describing an object) but the contents may and the form (for example the font) may be different.

The default definition of displayString in Object is simply printString and most classes don't redefine this definition.

To follow the spirit of the distinction of the two messages, we will expect them to produce different strings. Message printString will produce text such as

```
'Item: Apple, a Currency: 13 dollars, 27 cents'
```

taking advantage of printString in Currency, whereas displayString will produce

```
sugar    price: 3 dollars  15 cents
```

Both methods will be defined in class Item. The definition of displayItems is

displayItems

"Display alphabetically all items selected by the customer along with their prices. Use alphabetical order."

Transcript clear.

```
(items asSet asSortedCollection: [:item1 :item2 | item1 name < item2 name])
do: [:item | Transcript show: item displayString; cr]
```

The notable thing about this definition is that the conversion to a sorted collection uses a special sort block instead of the simple asSortedCollection message. The definition of displayExpensivelItems is

displayExpensivelItems

"Display a heading and all items that cost more than 10 dollars."

Transcript cr; cr; show: 'Expensive items: '; cr.

```
items do: [:item | item isExpensive ifTrue: [Transcript show: item displayString; cr]]
```

We leave the other methods in ShoppingMinder to you as an exercise.

Class Item

The name:price: creation method that we used in addItem expects a String and a Number which it converts to a Currency object. Its definition is as follows:

name: aString price: aNumber

"Create new instance using aString for name and a aFixedPoint for currency."

| dollars cents |

dollars := aNumber asInteger.

"Get the integer part of the price."

cents := ((aNumber - dollars) * 100) asInteger.

"Get the fractional part."

^self new

name: aString;

price: (Currency dollars: dollars cents: cents)

where we assume that accessing messages name: and price: have already been defined. The definition of displayString is as follows:

displayString

```
| tabString |
tabString := String with: Character tab.
^name , tabString , 'price: ' , price dollars printString , ' dollars ' ,
(price cents rem: 100) printString , ' cents'
```

We leave the rest, including the `printString` method, as an exercise. By the way, we are not happy with the strategy that we used to implement our application. It would have been better to implement the domain class first and test it, and then go to the application model and test it with the already existing domain class. Most developers proceed in this order.

What happens when a collection grows?

When you add a new element to a collection filled to capacity, the following steps take place:

1. A new collection of the same kind as the receiver is created with double the capacity of the original.
2. All elements of the receiver are copied to the new collection. (In reality, only the pointers are copied, of course.)
3. All references from the system to the receiver are switched to point to the new collection (Figure 8.4). This step uses message `become`: which is understood by all objects and may be time consuming.
4. Since there are now no references to the old collection, it is garbage collected².

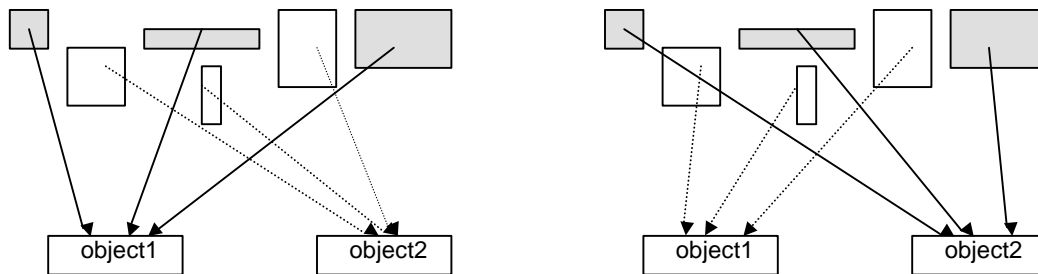


Figure 8.4. Message `object1 become: object2` switches references.

The heart of the process is the following method:

changeCapacityTo: newCapacity

```
"Change the capacity of the collection to be newCapacity."
| newSelf |
"Make an empty copy whose capacity is obtained elsewhere."
newSelf := self copyEmpty: newCapacity.
"Initialize value of firstIndex."
newSelf setIndicesFrom: 1.
"Add all elements of existing collection fast - with no checks."
firstIndex to: lastIndex do:
    [ :index | newSelf addLastNoCheck: (self basicAt: index)].
"Switch references."
self become: newSelf
```

The nature of collections - variable sized classes

The definition of `OrderedCollection` in the browser

```
SequenceableCollection variableSubclass: #OrderedCollection
instanceVariableNames: 'firstIndex lastIndex '
```

² See Appendix 8 for a discussion of garbage collection.

```
classVariableNames: "  
poolDictionaries: "  
category: 'Collections-Sequenceable'
```

uses a creation message different from most other classes such as

```
Object subclass: #Document  
instanceVariableNames: 'entities currentMargins '  
classVariableNames: 'DefaultMargins DefaultTabStops '  
poolDictionaries: "  
category: 'System-Printing'
```

You will find that several other collection class definitions use the keyword *variableSubclass:* instead of *subclass:*. This is because Smalltalk recognizes essentially two types of classes - those whose components are accessed by an index, and those that don't have index-based access. Both types of classes may or may not have named instance variables. The standard browser template for declaring a new class assumes the second kind of class because most classes have only named variables, and if you want to define a class with indexed variables (a *variable* class), you must modify the definition template in the browser.

A class such as *OrderedCollection* thus has the structure shown in Figure 8.5 and contains *unnamed* index elements and (possibly) *named* instance variables. Named instance variables are accessed via an accessing method, but indexed elements are accessed using a mechanism such as *self at: index* and *self at: index put: object*. The fact that there is an indexed component can only be recognized from the definition of the class because the indexed components don't have any variable name associated with them.

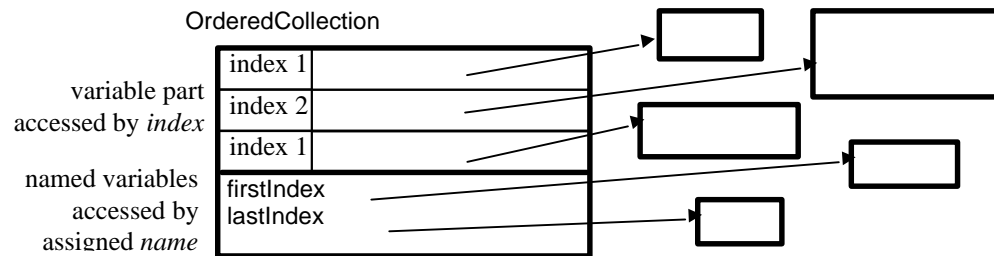


Figure 8.5. Indexed elements are accessed by index as in *self at: index* rather than by name.

Main lessons learned:

- VisualWorks has two forms of messages converting an object to a *String*. Message *printString* should be used for debugging, *displayString* is for output to the user interface.
- Smalltalk recognizes two main types of classes - those that have index-accessed elements and those that don't. The standard browser template for creating a new class creates a class without access by index unless the superclass is a variable size class itself. To create a new variable-size class that is not a subclass of a variable-size class, modify the class creation template.

Exercises

1. Complete the shopping minder example. Add *printString* to both new classes for testing.
2. Write a code fragment to request a list of labels from the user and use them in a multiple choice dialog. Repeat with both forms of *Dialog*'s multiple choice messages.
3. Read Appendix 3 and write an expression to find all variable class in the library.
4. *ClassOrderedCollection* defines method *removeAllSuchThat: aBlock*. Compare it with *reject:*.

8.3 Ordered collections as the basis of dependency

In this section, we will take a closer look at an important application of ordered collection - the dependency mechanism - and illustrate it on an example. Let us start by recapitulating what we know about dependency.

Any object may have any number of dependents. The basic dependency mechanism is defined in class `Object` and redefined in class `Model`. `Model`'s implementation is preferable because it holds dependents in instance variable called `dependents` (an `OrderedCollection`) rather than a global variable as `Object` does. As a consequence, when a `Model` object ceases to exist, its dependents are no longer referenced and may be garbage collected, whereas `Object`'s dependents remain referenced until explicitly removed from the global variable that references them. If `Object`'s dependents are forgotten, they will thus stay in the image forever.

The protocol for adding new dependents and removing existing ones is defined in class `Object` and only the accessing method for the `dependents` collection is redefined in `Model`³. The dependency mechanism is based on three changed methods called `changed`, `changed:`, and `changed:with:`. All three are defined in class `Object`, exhibit the same basic behavior (Figure 8.6), and differ only in their number of arguments.

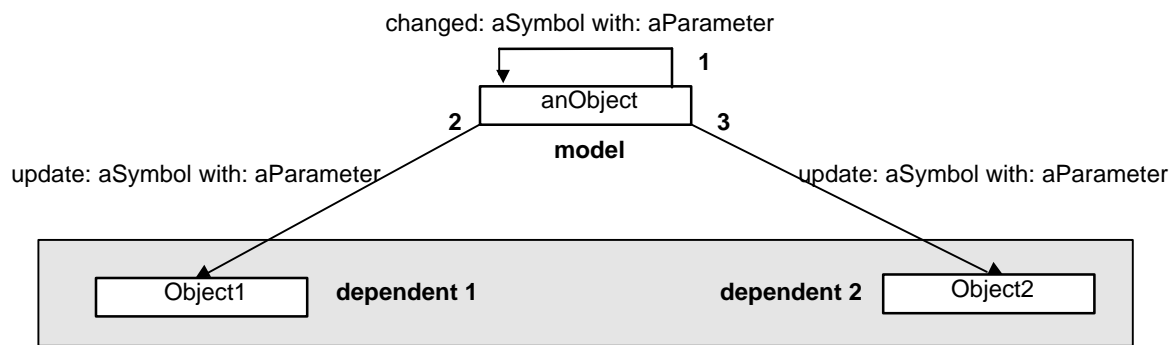


Figure 8.6. A model with two dependents and its response to self `changed:with:`. Numbers indicate order of message execution.

Dependency works as follows: When the model object needs to notify its dependents that it has changed in a way that might affect them, it sends *itself* an appropriate changed message as in

self changed

or

self changed: #value

or

self changed: #value with: 153

In response to the last message, for example, the definition of changed messages in `Object` then sends `update: #value with: 153` to each of its dependents:

changed: anAspectSymbol with: aParameter

"The receiver changed. The change is denoted by the argument `anAspectSymbol`. Usually the argument is a `Symbol` that is part of the dependent's change protocol, that is, some aspect of the object's behavior, and `aParameter` is additional information. Inform all of the dependents."

self myDependents update: anAspectSymbol with: aParameter from: self

³ The ease with which `Model` can redefine its inherited behavior by redefining only one accessing method illustrates the advantage of using accessor methods instead of direct access to instance and class variables.

where `myDependents` returns the `dependents` collection. Each dependent then responds according to its own definition of `update:with:`. Typically, the definition of `update:with:` examines the argument `anAspectSymbol`, decides whether it cares to do anything for the kind of change indicated by the argument and if it does, it uses `aParameter` to execute the appropriate response. The response of individual dependents may range from identical for each dependent to different for each dependent, to none for some or all dependents. A default implementation of all update messages is defined in `Object` and does not do anything. As a consequence, if a dependent does not have an update method, it simply does not do anything when the model changes (unless `update` is defined in a superclass).

The implementation of changed messages is interesting. The simplest one - the `changed` method with no arguments - is as follows:

```
changed  
self changed: nil
```

This definition thus does not send any update message and depends on `changed:` to do so. The `changed:` message is defined as

```
changed: anAspectSymbol  
self changed: anAspectSymbol with: nil
```

It again passes the real work to a more complete version of `changed` which is defined as follows:

```
changed: anAspectSymbol with: aParameter  
self myDependents update: anAspectSymbol with: aParameter from: self
```

This method finally sends an `update` message to the dependents. This update message contains an extra argument - the sender of the message. This is essential in some situations because it allows the dependent to ask the model for additional information.

We conclude that no matter which `changed` message the model sends, the result is always the same update message, namely `update:with:from:`. The question now arises how a dependent that does not define this rich version of `update` responds. As an example, how does a dependent who defines only `update: aSymbol` respond to `update:with:from:`?

The answer is obvious: If an object gets a message that it does not understand, Smalltalk looks for the method in the method dictionary of its superclass. If this class does not contain the definition, it searches the next superclass, and so on until it reaches class `Object`. To find how the update mechanism works, we must thus look at the definition of `update:` in `Object` where we find, for example,

```
update: anAspectSymbol with: aParameter from: aSender
```

```
"Receive a change notice from an object, denoted by aSender, of whom the receiver is a dependent. The argument anAspectSymbol is typically a Symbol that indicates what change has occurred and aParameter is additional information. The default behavior defined here is to do nothing; a subclass might want to change itself in some way. Note that this implementation assumes that the object does not respond to this protocol and an attempt is made to try a simpler message."
```

```
^self update: anAspectSymbol with: aParameter
```

This shows that if the dependent does not understand the 'rich' update message, `Object` downgrades the message to a simpler update and asks the dependent to execute it. If the dependent does not have that definition, `update` again goes up the superclass chain and if it reaches `Object`, it is downgraded again, and so on, until either a suitable update definition is found in the receiver's class or its superclass or until execution eventually passes to `update` in `Object` - which does nothing.

We have already seen the most conspicuous use of dependency in `VisualWorks` in its implementation of the user interface: A widget is a dependent of its `ValueHolder` (a subclass of `Model`) and when the value holder gets a `value:` message, it informs its dependent widget via `changed` and `update`, and the widget then gets the necessary information from the `ValueHolder` and redraws itself.

Although UI implementation is the most prominent use of dependency, there are other uses as we will illustrate in the following example. Note that this example (and all other uses of dependency) could be

implemented without the dependency mechanism. The use of dependency just makes the implementation more logical and neater because it gives the model object a uniform way to deal with all its dependents, whatever their nature may be.

Example. Dependence of object state on weather

The most common use of dependency is to send `changed: aValue` and use the value in the `update:` method of the dependent to decide exactly what to do. We will now present a very simple example that uses this form of dependency.

Problem: Objects on Earth respond to weather in various ways. When the sun is out, people sit in the shade, talk, and smile, birds sing, and stones get warm. When it rains, people get umbrellas, stones get wet, and birds don't do anything. When it snows, people ski, and birds and stones disappear. We will now write a program that asks the user to select current weather (Figure 8.7) and prints the state of people, birds, and stones in the Transcript. The program terminates when the user clicks *Stop*.

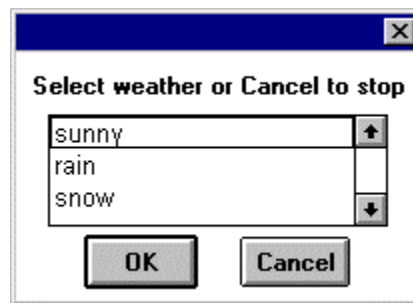


Figure 8.7. User interface for text example.

Solution. We will define a class for each part of the problem - Weather, Person, Bird, and Stone. Class Weather will be responsible for running the program and displaying the user interface, Person, Bird, and Stone will be its dependents and will respond to weather changes by printing their state to the Transcript.

Class Person

The only responsibility of Person is to respond to an `update` message. This message must specify the new weather, a Symbol, one of `#sunny`, `#rainy`, or `#snow`. This is the 'aspect' of change and we will thus use the `update: anAspect` message. According to the specification, the definition is simply

update: aSymbol

```
"Display appropriate string according to weather described by aSymbol."  
(aSymbol == #sunny)  
    ifTrue: [^Transcript show: 'People talk and sit in the shade.' ; cr].  
(aSymbol == #rainy)  
    ifTrue: [^Transcript show: 'People get umbrellas.' ; cr].  
(aSymbol == #snow)      "This test is not really necessary."  
    ifTrue: [^Transcript show: 'People go skiing.' ; cr].
```

No other methods need to be defined because instances can be created with the new message.

Class Bird

This class is essentially identical to Person but the details of `update:` are different because birds behave differently:

update: aSymbol

```
"Display appropriate string according to weather described by aSymbol."  
(aSymbol == #sunny)
```



```
ifTrue: [^Transcript show: 'Birds sing.' ; cr].  
(aSymbol == #snow)  
fTrue: [^Transcript show: 'Birds disappear.' ; cr].
```

Note that Bird does not respond when the value of the update: argument is #rainy. No other methods need to be defined because instances can be created with the new message.

Class Stone

Similar to the two classes above and left as an exercise.

Class Weather.

Since Weather has dependents it will be a subclass of Model. Since it is also an application model, we will make it a subclass of ApplicationModel (which is a subclass of Model). Its initialization method must assign new instances of Person, Bird, and Stone as its dependents:

initialize

"Create new instance and add Person, Bird, and Stone as its dependents."

self dependents

addDependent: Person new;

addDependent: Bird new;

addDependent: Stone new.

self run "We don't know how to create lists yet so we will use this intermediate implementation for now."

and the instance method run is

run

"Repeatedly ask user to select weather and stop when the selection is Cancel. Notify dependents."

repeat [selection := Dialog choose: 'Select weather or Cancel to stop'

fromList: #('sunny' 'rain' 'snow')

values: #(#sunny #rainy #snow)

lines: 3

cancel: [].

selection isNil ifTrue: [^self].

self changed: selection]

Now assume that we decided to add another object, such as a river. We must define its class and modify the initialize method to add a River dependent, but the handling of changes does not change because the Model does not need to be explicitly aware of its individual dependents and does not care what they are and what are the details of their behavior as long as they know how to respond to an appropriate update message. This is dependency's main claim to fame.

Main lessons learned:

- Dependency is defined by specifying the dependents of a model object.
- The model object is normally a direct or indirect subclass of Model although any object can have dependents via the mechanism defined in Object.
- Dependents in Model subclasses are stored in an ordered collection.
- When the model changes in a way that may affect its dependents, it sends itself one of the changed messages.
- The changed message sends a matching update message to each dependent.
- There are several varieties of changed and update and they differ in the number of keywords.
- The update method defined in the dependent's classes does not have to match the changed message sent by the model but it normally does.
- Dependency is used mainly by the user interface but there are many other possibilities for its use.
- All problems can be solved without dependency but the use of dependency makes the solution cleaner

and more flexible.

Exercises

1. Browse and describe implementation of dependency in class Object.
2. Complete the weather example and test it.
3. Add a river object to the example. When it is sunny, the river gets warm, when it snows, the river freezes. Rain does not have any effect.

8.4. Tennis - another example of dependency

Have you ever watched the spectators at a tennis game? As the ball moves across the court, their heads turn in unison as if enchanted. In this example, we want to simulate this effect. Consider a very simple implementation of the problem, assuming a collection of spectators arranged in a straight line in parallel with the axis of the court and a ball moving along the axis from one end of the court to the other (Figure 8.8).

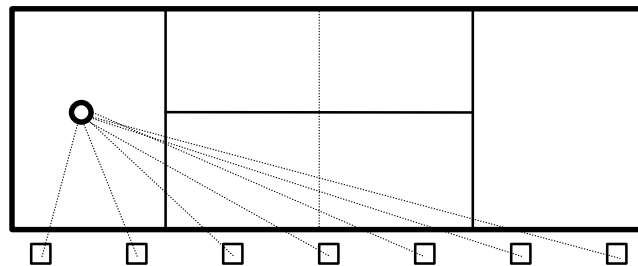


Figure 8.8. Tennis court with spectators following the ball's motion.

In this section, we will not represent the court and the moving ball graphically because we don't know how to do that yet; we refer you to Chapter 12 for information necessary for a graphical implementation. Our implementation will simply print the position of the ball and spectator information to the Transcript in the following form:

```
Move number 1
Ball position: 10 direction: left.
Spectator location: 1 angle: 1.25
Spectator location: 4 angle: 1.11
Spectator location: 7 angle: 0.79
```

```
Move number 2
Ball position: 9 direction: left.
Spectator location: 1 angle: 1.21
Spectator location: 4 angle: 1.03
Spectator location: 7 angle: 0.59
```

and so on for each consecutive ball displacement. Angles are expressed in radians.

To make the program more interesting, we will provide the user interface in Figure 8.9 allowing selection of court parameters, the number of spectators, and the number of successive changes of ball positions (*Number of moves*). The location of spectators along the court will be calculated from the assumption that spectators are uniformly distributed.

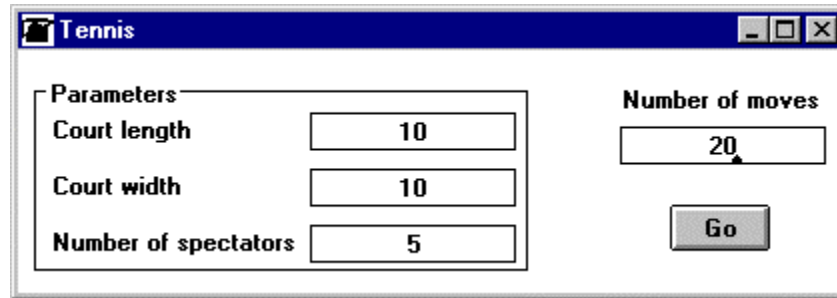


Figure 8.9. User interface for Tennis program with initial default parameters.

Specification

The statement of the problem is clear and there are only two interesting scenarios - opening the application and clicking *Go*.

Scenario 1: Starting the program

1. *User* executes expression such as *Tennis* open
2. *Program* opens the window in Figure 8.7.

Scenario 2: Starting simulation by clicking *Go*.

1. User clicks *Go*.
2. Program places the ball at the right end of the court and starts simulation, performing a series of ball displacements accompanied by adjustments of spectators' viewing angles.

Preliminary design

The specification suggests three classes - *Tennis*, *Ball*, and *Spectator*. Their initial description is as follows:

Tennis: I maintain simulation parameters, perform the simulation, and display the user interface.

Responsibilities

- interface specs - define UI window Collaborators
- creation - open user interface, initialize default values Ball, Spectator
- action - response to *Go*. Reset ball and perform the required number of steps, printing simulation results in Transcript Ball, Spectator

Ball: I represent the tennis ball moving across the tennis court.

Responsibilities

- creation - create a ball with initial position and direction Collaborators
- moving - calculate new position and notify spectators Spectator

Spectator: I represent a spectator who turns his or her head while watching the moving ball.

Responsibilities

- creation - create a spectator at given location Collaborators
- watching - calculate new head position Ball

We will now check whether these three candidate classes can support our two scenarios and how.

Scenario 1: Starting the program

1. *User* executes expression *Tennis* open
2. *Tennis* initializes user interface parameters and opens the interface in Figure 8.7.

Scenario 2: Starting simulation by clicking *Go*.

1. User clicks *Go*.
2. Tennis creates a new **Ball** and a new collection of **Spectator** objects according to the values of user interface parameters.
3. Tennis executes simulation by asking the ball to move the specified number of times, displaying the required output for each step. Ball moves by incrementing or decrementing its position depending on its current direction of motion, notifies all spectators of its new position, and checks whether it has reached the end of the court and should change direction. Each **Spectator** calculates its viewing angle from its location and the current position of the ball.

We conclude that our three classes should be able to handle the problem.

Design refinement

Let's start with a preliminary discussion of the place of the three classes in the class hierarchy. Tennis is a subclass of **ApplicationModel** because it is responsible for the user interface, **Ball** and **Spectator** have no class relatives and this suggests superclass **Object**.

As the next step, we will take a closer look at class descriptions starting with **Ball** and **Spectator**. Clearly, the state of **Spectator** depends directly on the position of the ball: Whenever the ball moves, every spectator is notified and responds by changing its viewing angle. Spectators are thus dependents of the ball. This has several consequences. First, **Ball** should thus be a subclass of **Model**. Second, the action responsibility of Tennis now requires only the collaboration of **Ball**, because **Ball** will take care of **Spectator**. With this insight, we can now rewrite the class descriptions as shown below. Note that we changed the descriptions of some of the responsibilities to reflect our better understanding of the situation.

Tennis: I perform the simulation and display the user interface.

Concrete class.

Superclass: **ApplicationModel**

Components: ball (**Ball**), spectators (**Array**), and aspect variables on integer values courtWidth, courtLength, numberOfMoves, and numberOfSpectators

Responsibilities

- | | |
|--|---|
| <ul style="list-style-type: none"> • interface specs - define UI window • creation - open user interface, initialize default values <ul style="list-style-type: none"> • initialize - initialize aspect variables, create a Ball • action - response to <i>Go</i>. <ul style="list-style-type: none"> • go - reset everything, create spectators and make them dependents of ball, perform the required number of steps, printing simulation steps in Transcript • printing - supply your description <ul style="list-style-type: none"> • printOn: - return descriptive string according to output specification | <p>Collaborators</p> <p>Ball</p> <p>Ball, Spectator</p> |
|--|---|

Ball: I represent the tennis ball moving back and forth across the tennis court.

Concrete class.

Superclass: Model

Components: court (Tennis), direction (Symbol) direction of motion - #left or #right, position (Integer) - rightmost position is 0, increases towards left

Responsibilities

Collaborators

- creation - create a ball
 - onCourt: aTennis - Ball needs access to Tennis to get simulation parameters at the start of each run
- moving - move the ball by one step
 - move - increment or decrement position depending on direction, notify dependent spectators, check for end of court and change direction if necessary
- printing - supply your description
 - printOn: - return descriptive string according to output specification

Spectator: I represent a spectator turning his or her head while watching the moving ball.

Concrete class.

Superclass: Object

Components: angle (Float) - viewing angle, distance (Integer) distance to ball line (court axis), location (Integer) - location measured from the right end of the court

Responsibilities

Collaborators

- creation - create a spectator
 - distance:location: - create new spectator at given distance from ball line and at given location from right end of court
- updating - respond to update request from Ball
 - update:with: - calculate new viewing angle from ball position
- printing - supply your description
 - printOn: - return descriptive string according to output specification

Figure 8.10 shows the Hierarchy Diagram and the Object Model Diagram.

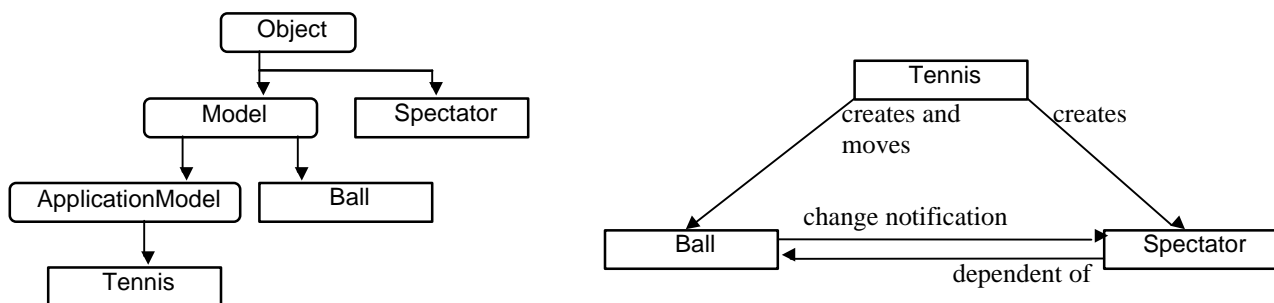


Figure 8.10. Hierarchy Diagram (left) and Object Model Diagram (right) of Tennis.

Implementation

We will now show some selected methods class by class.

Class Tennis

The definition of the class is

ApplicationModel subclass: #Tennis
instanceVariableNames: 'ball spectators courtWidth courtLength numberOfMoves

```
numberOfSpectators '  
classVariableNames: "  
poolDictionaries: "  
category: 'Tennis'
```

Initialization defines default values for value holders and creates a Ball:

initialize

```
"Initialize the opening values of simulation parameters, create a ball."  
courtLength := 10 asValue.  
courtWidth := 10 asValue.  
numberOfSpectators := 5 asValue.  
numberOfMoves := 20 asValue.  
ball := Ball for: self "Create tennis ball on this court."
```

The basis of the operation of the whole program is response to the *Go* button which is defined as follows:

go

```
"Reinitialize parameters according to current settings, execute the simulation the required number of times,  
and display the result."  
self reset.  
Transcript clear.  
1 to: numberOfMoves value  
do: [:number | Transcript cr; cr; show: 'Move number ', number printString; cr;  
show: self printString.  
self ball move]
```

Method *reset* reads the current settings of user interface value holders and uses them to create spectators and assign them to ball as its dependents:

reset

```
"Remove existing spectators, if any, create equally spaced new spectators and make them dependents of  
the ball. Used to update parameters before the start of each simulation run."  
| distance |  
1 to: spectators size do: [:index | self ball removeDependent: (spectators at: index)].  
spectators := Array new: self numberOfSpectators value.  
distance := (courtLength value / spectators size) truncated.  
1 to: spectators size do: [:index | spectators at: index put:  
(Spectator distance: distance location: 1 + (distance * (index - 1)))].  
spectators do: [:aSpectator | self ball addDependent: aSpectator]
```

Note that we had to remove existing spectators before creating new ones; we assume that simulation parameters have changed. Printing of results uses the *printOn:* method because this is only a test. The definitive user interface will be implemented when we learn about graphics.

printOn: aStream

```
"Display ball and spectator information."  
aStream nextPutAll: self ball printString , ' '; cr.  
self ball dependents do: [:aSpectator | aStream nextPutAll: aSpectator printString , ' '; cr]
```

The operation of this method depends on *printOn:* methods defined in *Ball* and *Spectator*.

Class Ball

The definition of the class is

Model subclass: #Ball

```
instanceVariableNames: 'court direction position '  
classVariableNames: "  
poolDictionaries: "  
category: 'Tennis'
```

Operation of **Ball** requires access to **Tennis** to obtain court parameters and we thus define the creation message as follows:

for: aTennis

```
"Create a ball at the right end of the court."  
  ^ (self new) court: aTennis; reset
```

Initialization depends on method **reset** which sets the ball's initial position and direction of motion:

reset

```
"Assign initial position (right end of court) and direction of motion (left)."  
  self position: self court courtLength value.  
  self direction: #left
```

The essence of the whole simulation is method **move** which moves the ball by one step and changes its direction when it reaches the end of the court:

move

```
"Make a one-pixel move in the current direction. Check for end of court and change direction if necessary."  
  self direction == #left  
    ifTrue: [self position: self position - 1]  
    ifFalse: [self position: self position + 1].  
  self changed: #position with: self position.      "This triggers the dependency mechanism."  
  self endOfCourt ifTrue: [self reverseDirection]
```

where the test for the end of the court is

endOfTrack

```
"Did I reach the end of the court? Return true or false."  
  ^ position = 1 or: [position = self court courtLength value]
```

and the change of direction is implemented as follows:

reverseDirection

```
"I reached the end of the course and direction of motion must be reversed."  
  self direction: (self direction == #left  
    ifTrue: [#right]  
    ifFalse: [#left])
```

Finally, printing uses the **printOn:** method which is defined as follows:

printOn: aStream

```
aStream nextPutAll: 'Ball position: ', self position printString, ' direction: ', self direction asString
```

Class Spectator

The definition of the class is

```
Object subclass: #Spectator  
  instanceVariableNames: 'angle distance location '  
  classVariableNames: "  
  poolDictionaries: "  
  category: 'Tennis'
```

To create a **Spectator**, we must initialize its distance from the right end of the court and its distance from the axis of ball motion:

distance: distanceInteger location: locationInteger

^self new distance: distanceInteger location: locationInteger

This definition depends on the following creation method defined on the instance side of Spectator protocol:

distance: distanceInteger location: locationInteger

"Complete initialization and calculate initial head angle for ball in position 1."

self distance: distanceInteger.
self location: locationInteger.
self calculateAngleFor: 1

Angle calculation uses some simple trigonometry and its definition is

calculateAngleFor: anInteger

"Find angle for this spectator from current ball position anInteger."

| xDistance |
xDistance := anInteger - location.
self angle: (xDistance / self distance) arcTan

The temporary variable xDistance is used only to make the code more readable.

Since spectators are dependents of Ball, they must understand an update message. The only parameter we need from the model is the ball's current position but to follow established conventions, the update: argument is a Symbol even though the method does not use it, and the second argument will be the ball's position:

update: aSymbol with: anInteger

"The ball has moved to position anInteger - recalculate head angle."

self calculateAngleFor: anInteger

Finally, the printOn: method is as follows:

printOn: aStream

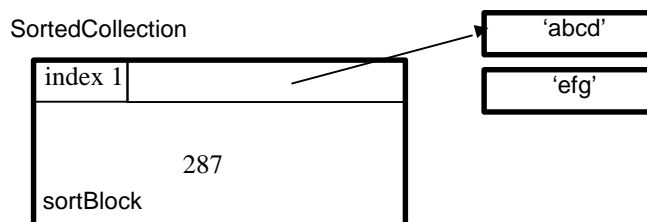
aStream nextPutAll: 'Spectator location: ', self location printString , ' angle: ',
(self angle asFixedPoint: 2) printString

Exercises

1. Complete and test the program from this section.
2. Dependency may be mutual as in the following simulation problem: Consider a collection of 5 nuclear particles distributed uniformly along a line. Each particle initially has some random amount of energy between 1 and 5. In each simulation step, the energy of a particle grows by a random integer amount between 1 and 3. When a particle's energy reaches or exceeds 10, it explodes, loses all its energy, and transfers it to the other particles in equal amounts. Implement the problem with a user interface allowing the user to control simulation parameters.

8.5 Class SortedCollection

SortedCollection is a subclass of OrderedCollection that automatically inserts new elements in their proper place among elements already in the collection. Insertion is based on a *sort block* stored in instance variable sortBlock (Figure 8.11). Most methods inherited from OrderedCollection work for sorted collections as well but some, such as addAfter:, are invalid because sorted collections determine where to put new elements themselves.



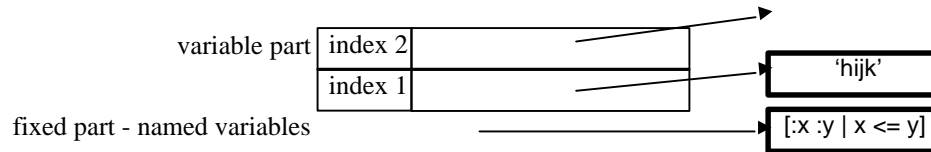


Figure 8.11. SortedCollection is a variable-size collection with one named instance variable whose value is a two-argument block. Its default value is `[:x :y | x <= y]`.

Since each SortedCollection keeps its sort block in instance variable `sortBlock`, its value can be assigned when the SortedCollection is being created or even during its lifetime. Most sorted collections are created with `new` or `new:`, both of which assign `sortBlock` the default value stored in class variable `DefaultSortBlock`. *This* value is assigned by the SortedCollection `initialize` method which is

initialize

```
DefaultSortBlock := [:x :y | x <= y]
```

This definition deserves several comments:

- None of the references to SortedCollection sends message `initialize`. What is its purpose then? This is executed to initialize SortedCollection when the class is first defined and the result is saved in the Smalltalk dictionary as a part of SortedCollection. You can change the default sort block by redefining `initialize` and re-executing SortedCollection `initialize` or by assigning a new value to `DefaultSortBlock` directly, and you could make the change permanent by saving the image. This, however, would change the effect of those existing uses of sorted collections that assume the current value of the sort block.
- The definition of `initialize` shows that default comparison uses message `<=`. This means that if you create a sorted collection with its default sort block, all elements that will ever be added to the collection must be able to compare themselves with all objects already in the collection using `<=`. Collections of strings and collections of numbers satisfy this condition. We can also define `<=` for other objects and compare, for example, rectangles by their size, Smalltalk methods by the amount of time that they take to execute, colors by the value of their hue or redness, and so on, and all these objects will now be able to sort themselves. The mechanism is very powerful.

The assignment of the default sort block in `new:` happens as follows:

new: anInteger

```
^(super new: anInteger) initialize
```

Here super `new:` refers to `new:` in OrderedCollection and creates a new uninitialized instance of SortedCollection. This SortedCollection object then executes the instance message `initialize` whose definition is

initialize

```
sortBlock := DefaultSortBlock
```

"Instance method accessing a class variable."

We have already mentioned that you can assign a different `sortBlock` when creating an instance of SortedCollection or during its lifetime. In the second case, the sorted collection automatically resorts itself according to the new sort block. The following code fragment shows both possibilities. It first creates a sorted collection with a non-default sort block and then changes the sort block to the default value:

```
| sortedCollection |
sortedCollection := SortedCollection sortBlock: [:x :y | x > y].    "Non-default sort block - descending order."
sortedCollection addAll: #(13 53 56 21 98 73 56 89).               "Elements added one by one and sorted."
Transcript clear; show: sortedCollection printString; cr.
sortedCollection sortBlock: [:x :y | x <= y].                     "Switch to a different sort block and resort."
Transcript show: sortedCollection printString
```

Execution produces

```
SortedCollection (98 89 73 56 56 53 21 13)
SortedCollection (13 21 53 56 56 73 89 98)
```

Although the code sends `sortBlock:` twice, the two messages are different: The first is a *class* method and creates a `SortedCollection` with a non-default sort block, the second is an *instance* method and changes the sort block of the existing `SortedCollection` and resorts it.

Example 1. Sorting a collection of random numbers

Problem: Generate ten random numbers, display them in the Transcript in the order in which they were generated, and then again in the ascending order of their magnitudes. Print the numbers in the format `#.###` where `#` is a digit. An example of the desired output is

```
0.030 0.762 0.421 0.988 0.299 0.048 0.786 0.921 0.718 0.300
0.030 0.048 0.299 0.300 0.421 0.718 0.762 0.786 0.921 0.988
```

Solution: The principle is simple:

1. Create a `SortedCollection` with the default sort block.
2. Create a random number generator.
3. Send `next` to the random number generator ten times and each time display the generated number and add it to the `SortedCollection`.
4. Display the resulting `SortedCollection` in the desired format.

The problem of formatting the output is new. There may already be a method to do this and we thus search the library using the *Implementers Of ...* browser for something like `print...` We find many matching selectors and one of them is called `print:formattedBy:` which seems promising. It is defined as a class method in `PrintConverter` and the definition shows that it indeed does what we want. As an example,

```
PrintConverter print: 3.14567 formattedBy: '#.###'
```

returns the string `'3.14'`. With this information, we can solve our problem as follows:

```
| generator number randomNumbers |
"Create a sorted collection and a random number generator."
randomNumbers := SortedCollection new: 10.      "Use default sort block."
generator := Random new.
Transcript clear.
"Generate random numbers, print them, and save in sorted collection."
10 timesRepeat: [number := generator next.
                  Transcript show: (PrintConverter print: number formattedBy: '#.###'), ' '.
                  randomNumbers add: number].

Transcript cr.
"Display all elements of the collection in the desired format."
randomNumbers do: [:element| Transcript show: (PrintConverter print: element formattedBy: '#.###'), ' ']
```

We ran the program several times getting a different output each time as expected.

Example 2: Sorting shopping minder items.

Problem: We are to modify the shopping minder from the previous section to display the purchased items in alphabetical order.

Solution: We only need to use a `SortedCollection` in `initialize` instead of an `OrderedCollection` to store the items, and the sort block must make comparisons on the basis of the name of the item:

initialize

```
"Initialize items to a suitably sized sorted collection. Sort alphabetically by name."  
items := SortedCollection sortBlock: [:item1 :item2] (item1 name) < (item2 name)].  
self execute
```

Main lessons learned:

- Sorted collections are ordered collections that automatically sort their elements according to a sort block.
- The default sort block uses <= but a different sort block can be assigned by a class or an instance message.
- As elements are added to a sorted collection, they are automatically inserted into the proper place according to the sort block.
- Every pair of elements must be able to execute the sort block.

Exercises

1. Find and examine all references to sortBlock: and determine which of them are instance messages: and why they are used.
2. Study and describe how SortedCollection adds a new element.
3. Browse SortedCollectionWithPolicy and explain how it differs from SortedCollection.
4. Formulate the sort block for storing entries of the library catalog from the Section 8.3 in a SortedCollection using the alphabetical order of author names. Make up any necessary accessing methods.
5. Write a code fragment to create ten random rectangles and sort them in the order of increasing area.
6. Items are frequently sorted by more than one property. As an example, listings in telephone directories are by last name, and if last names are the same then by first name. Extend the previous exercise by using the area as the primary comparison key, and the circumference as the second comparison key, used when areas are the same.
7. Write a program to let the user compare the speed of execution of selected unary mathematical messages such as squared, ln, sin, and cos. Message names are entered by the user. Use SortedCollection.
8. Find forms of perform: that can be used when the method has one or more arguments and use them to write a more general version of the previous exercise that accepts any mathematical messages. (Hint: Use method keywords to obtain the number of keywords in the selector entered by the user.)
9. Modify the shopping minder to sort items in decreasing order of price.
10. Modify the System Browser to display names of categories in alphabetical order.
11. Ask the user to enter a string pattern and a series of strings, and sort the strings by their similarity to the pattern. As an example, if the user enters pattern 'abcd' and then strings 'abcd', 'abbd', 'aaad', and 'xxxx', one would expect that ordering by similarity will produce 'abcd', 'abbd', 'aaad', and 'xxxx'. (Hint: VisualWorks has a method for measuring the similarity of two strings. It is used by command *correct* displayed when you misspell the name of a variable. Find it.)
12. Ask the user to enter several labels and display them in alphabetical order in a multiple choice dialog.

8.6 The List collection

Class List combines properties of ordered and sorted collections and adds dependency. According to the manual, List is intended to supersede ordered and sorted collections but its main use in the library is as an item holder in multi-item GUI widgets. This is consistent with the placement of List in the UIBasics - Collections category in the library.

In terms of implementation, List is not on the OrderedCollection branch of the Collection subtree, but a subclass of ArrayedCollection. Its instance variables are collection, limit, collectionSize, and dependents.

According to the class comment, collection holds the list's elements and the creation methods initializes it to an Array. This means that List is not itself a collection but contains a collection as one of its components. The role of limit is to hold the index of the last element in the collection that holds a valid element. This is because List makes a distinction between size and capacity and must keep track of how many of its slots are valid elements. collectionSize holds the size of the collection. dependents is nil, one object, or a collection of objects and implements the dependency mechanism: When an object is added or deleted, all dependents are notified.

Lists can be used as sorted collections because they can sort their elements. However, List does *not* have a sort block and does not automatically insert new elements. Instead, new elements are added using the adding protocol of OrderedCollection, for example addLast: and addFirst:. To sort a List, send it sortWith: aSortBlock or the sort message which uses the default sort block [:a :b | a <= b].

If List does not have a sort block, how can it sort itself? The answer is that a List does *not* sort itself but asks class SequenceableCollectionSorter to do it. The details are obvious from the following definitions of List's sorting methods:

sort

```
^SequenceableCollectionSorter sort: self
```

sortWith: aBlock

```
^SequenceableCollectionSorter sort: self using: aBlock
```

SequenceableCollectionSorter, a class that knows how to do sorting, is a nice example of object-oriented thinking - an object that can perform a service for its clients.

We will use List when we present widgets that depend on it later in this chapter.

Main lessons learned:

- List objects resemble ordered and sorted collections and add dependency handling.
- For accessing, lists behave like ordered collections.
- Unlike sorted collections, List does not sort itself automatically but must be explicitly asked to do so.
- A List sorts itself by sending a message to SequenceableCollectionSorter.
- The most common use of lists is as item holders of multi-item GUI widgets.

Exercises

1. List initializes collection to an Array which is a fixed size collection. What happens when you add a new item? What happens when you remove an element?
2. Write a program to test how sorting time depends on the number of elements in the list. The program should create a list with 10, 100, 1,000, and 10,000 random numbers and send them the sort message. Plot the numbers as a function of the size of the list.
3. Sorted collections insert a new element into its proper place as soon as it is added, lists sort their elements only when explicitly requested to do so. Write a program to compare the speed of the two approaches by creating a sorted collection containing a set of 100, 1,000, and 10,000 random numbers with a List and with a SortedCollection. Create the two collections so that they don't need to grow.

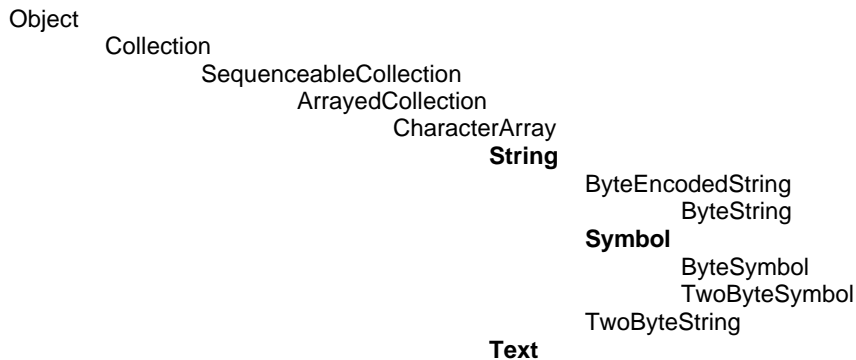
4. Class `SequenceableCollectionSorter` can be used with any sequenceable collection but its main purpose is to sort `List` objects, for example to display items in a multi-item widget in a particular order. Write a short description of its behavior and instance variables.
5. Write a comparison of `List` and `OrderedCollection` highlighting the similarities and differences in their behavior and implementation.
6. `List` is one of the few collection classes that are not variable-size classes and that store their elements in an instance variable. What does `List` gain from being in the `Collection` tree?

8.7 String, Text, and Symbol - an introduction

Classes `String`, `Text`, and `Symbol` are all in essence collections of characters such as letters, digits, and punctuation symbols. A `String` is a sequence of characters and its functionality includes searches, deletions, and replacements. Strings are concerned with contents and don't contain any information about rendering - font, size, boldness, and so on. When they are displayed ('rendered'), they use default system representation.

A `Symbol` is also a sequence of characters but the rules for its formation are slightly different and its main use is for selectors - names of methods; this is reflected in its different protocol. Unlike strings, symbols are unique in that only one instance of a particular sequence of characters can exist at a time. This is their most important defining property which guarantees very fast access. Finally, instances of `Text` are aggregate objects containing a string and information about its emphasis - font, size, color, and other rendering information.

The internal natures of `String` and `Symbol` are closely related - both are essentially collections of characters - whereas `Text` objects are related conceptually but their internal structure is different. This is reflected in the class hierarchy: Classes `String` and `Symbol` are on the same branch but `Text` is separate:



All three classes inherit many useful methods from the abstract class `CharacterArray`. We used strings and symbols informally before and we will now present some new information and examples of both. `Text` will be covered in the next section.

Class String

`String` is an *abstract class* that understands messages such as finding substrings and replacing substrings with other strings. Its superclass is the abstract superclass `CharacterArray` which means that its elements are characters (instances of class `Character`). *Concrete* strings are implemented by concrete subclasses of `String`, each designed for a different kind of character encoding. In most applications, strings are instances of `ByteString` which encodes its characters as one-byte ASCII codes. For large character sets requiring more codes, strings can be implemented as `TwoByteString` objects.

The fact that `String` is an abstract class and that string objects are instances of one of its concrete subclasses is hidden and you don't have to think about it. This is because when you create a new string as in

`String new: 20` "String with room for 20 characters"

the definition of new: in class String automatically determines which concrete subclass to use as follows:

new: size

^self defaultImplementor new: size

Here defaultImplementor is a class method defined as follows:

defaultImplementor

"Answer a class that usually represents strings."

^ByteString

As a consequence,

String new: 20

is equivalent to

ByteString new: 20

and all messages to the resulting string object are handled by class ByteString. This trick of using an abstract class as if it were concrete and allowing it to select the most suitable concrete class is used in several places and we will encounter it again later. Its advantage is that applications can be made independent of the concrete class and the same program may be, for example, executed on different platforms that use different defaults.

Most of the interesting protocol of String is defined in CharacterArray. Since String is a collection, it also inherits the rich collection protocol of all its superclasses up to Collection. The most useful methods defined in String include comparison and conversion to text via asText which allows strings to be used where Text objects are expected.

The protocols inherited from CharacterArray include accessing (replacement and search), converting (to number, text, symbol, and other related objects), copying, comparing (alphabetical relations), and displaying on the screen. One example of an accessing message is changeFrom: start to: stop with: replacement which takes the receiver string, removes characters indexed from start to stop, and inserts the string provided as the third argument starting at index start. The replacement string may contain fewer, more or the same number of characters as the specified range of indices and when start > stop, the argument string is simply inserted into the original. Another useful accessing message is findString: subString startingAt: start which searches for the first occurrence of subString starting from start and returns the index of its first character or 0 if the substring is not found. We will now develop a new method using both messages.

Example 1: Replacing substrings

Problem: Write a method with three string arguments: a start string, a match string, and a replacement string. The method finds all occurrences of the match substring in the start string and substitutes the replacement string for them. The result is returned.

As an example, if the start string is 'a secret 1 containing encoded sub1s' and the match string is '1', and the replacement string is 'string', the program should replace every occurrence of 1 with 'string' and produce 'a secret string containing substrings'.

Solution: The basis of the method is the following algorithm:

1. Set starting position to 1.
2. Look for the next occurrence of the match string. If a match is found, make the replacement and repeat this step.

The algorithm is easy to implement because all steps can be implemented with built-in messages. We will put the method in the *accessing* protocol in *CharacterArray* because it is related to other methods in this protocol and because it will be useful in subclasses. The definition is as follows:

replace: matchString with: replacementString

```
"Replace all occurrence of matchString in string with replacementString."  
| index newString matchSize |  
newString := self copy.  
matchSize := matchString size.  
"Find all occurrences and do replacement."  
[index := newString findString: matchString startingAt: 1. index = 0]  
whileFalse:  
    [newString changeFrom: index to: index + matchSize - 1 with: replacementString].  
^ newString
```

Unfortunately, when you try to execute the method, it enters into an infinite loop! The problem is due to our misunderstanding of the exact operation of *changeFrom:to:with:.* We leave it to you to find and correct this common error.

Closing comments

Before we close this very brief introduction to *String*, here is a useful piece of information. As you know, the most common way of concatenating two short strings (and many other collections) is with the, message as in

```
'abc' , 'def'      "Returns 'abcdef'."
```

However, what if you wanted to concatenate a string with a single character?

```
'abc' , $d
```

does not work because one cannot concatenate a string with a character. One could, of course, use

```
'abc' , 'd'
```

but what if the character is calculated or stored in a variable? In this case, you can use the *copyWith:* message from *SequenceableCollection* which creates a copy of a collection and appends the argument at the end as in

```
'abc' copyWith: aCharacter
```

This works because a string is a collection of characters.

Another comment related to concatenation is that the *,* method is very inefficient. When speed is important, concatenate strings using streams, a topic covered in Chapter 10. Since concatenation is usually an essential part of creating a string, streams are essential for *efficient* use of strings in general.

Finally, the concatenation method is defined in *SequenceableCollection* and its use is not limited to strings. As an example, you can execute

```
orderedCollection1 , orderedCollection2
```

Class Symbol

Symbol is a special kind of *String* whose literal form is a sequence of characters preceded by the *#* character as in *#aWidget* or *#aMethod* or even *#'abc def'*. A *Symbol* must not include characters such as blank space, tab, line feed or the *#* character itself, unless the string following *#* is a literal string. As an example, *#ab cd* is not a single *Symbol* but *#'ab cd'* is - and the apostrophes are *not* a part of it.

Unlike instances of *String*, *Symbols* are implemented as *unique* objects. This means that if you create two or more *Symbols* consisting of identical characters, they will be stored in memory only once. On

the other hand, if you create two *strings* consisting of identical characters, they will be stored as two separate objects. As a consequence, two different instances of *String* may be *equal* (message =) but not *equivalent* (message ==), whereas for symbols, equality and equivalence are the same:

'characters' = 'characters'	"Returns true."
'characters' == 'characters'	"Returns false."
#characters = #characters	"Returns true."
#characters == #characters	"Returns true."

Because Symbols are unique their comparison uses *equivalence* which, as we know, is fast because it requires only comparison of pointers to internal memory representations. Strings, on the other hand, are normally compared for equality which may be very time consuming. Note that although Symbol defines == to be the same as = and one could thus use = with the same effect, most programmers use == because it is faster, eliminating the = message send.

Symbols are used mainly for names of methods as in our comparison of speed of number messages, as widget IDs and *Action* and *Aspect* methods (see any *windowSpec* method for an example), and in similar situations in which uniqueness makes sense (such as days of the week, male and female gender, and other non-ambiguous characteristics) and in which the speed of comparison is important. Because one of their main uses is to represent method selectors, class *Symbol* contains a protocol dedicated to recognizing unary, binary, and keyword messages and similar aspects of selectors. The following example illustrates this protocol.

Example 2: Get information about messages

Problem: Write a program to count how many unary, binary, and keyword methods are defined in the instance protocols of *Object*. In addition, find how many of the keyword messages have one argument, how many have two arguments, and so on. Print the result in Transcript in the following format:

Total number of methods: 161

Number of unary methods: 81

Number of binary methods: 5

Number of keyword methods: 75

Number of arguments/Number of methods

1: 53
2: 15
3: 5
4: 2

Solution: The problem is quite specialized and we will implement it as a code fragment. The implementation will use the following algorithm:

1. Create an ordered collection *keywordMethods* to hold the selectors of all keyword methods.
2. Enumerate over all methods defined in *Object* as follows:
 - a. If the method is unary, increment the count of unary messages.
 - b. If the method is binary, increment the count of binary messages.
 - c. If the method is a keyword method, increment the count and add its selector to *keywordMethods*.
3. Enumerate over all selectors in *keywordMethods* and count the number of methods with one argument, two arguments, and so on. Store the number of keyword messages with one argument in element 1 of array *counts*, the number of keyword messages with two argument in element 2, and so on.
4. Create an array called *keywordCounts* whose size is equal to the maximum number of keywords in *keywordMethods*. Store the number of all keyword messages of size *n* in the *n*-th element of this array.
5. Print results.

To implement this algorithm, we must know how to find all methods defined in a class, how to recognize whether they are unary, binary, or keyword methods and determine how many arguments they have. As we have already mentioned, *Symbol* is mainly used for selectors and it contains the selector-related

methods. But how about finding all methods defined in a class? One way to find out is to think of a Smalltalk tool that performs a similar function, and as usual, this tool is the Browser; its class view <operate> menu includes command *find method* which lists all instance or class methods. Its implementation in class Browser must know how to find the methods! When you examine Browser, you will find that the following definition contains the desired information:

findMethodAndSelectAlphabetic

```
"Show a menu of the methods implemented by this class. Select the chosen one."  
| chosenSelector selectorCollection |  
self changeRequest ifFalse: [^self].  
(selectorCollection := self selectedClass organization elements asSortedCollection) size = 0  
etc. - the rest is irrelevant for our needs
```

It seems that the part organization elements does what we need. To test it, we execute

Object organization elements

with *inspect* and find that it returns an array containing all instance methods defined in Object⁴.

The next task is to find Symbol methods that recognize various types of selectors and number of arguments. We find that instance protocol *system primitives* contains methods *isInfix* (recognizes binary messages), *isKeyword* (recognizes keyword messages), and *numArgs* (returns the number of arguments of a keyword message). There is no special method for unary messages but unary messages are those that are not keyword or binary. With this information, we can now convert our algorithm into the following code:

```
|unaryCount binaryCount keywordCounts keywordMethods methods |  
unaryCount := 0.  
binaryCount := 0.  
keywordMethods := OrderedCollection new.  
  
"Enumerate over all selectors defined in Object and store their counts in unaryCount, binaryCount, and  
keywordCount."  
(methods := Object organization elements) do:  
    [:method| method isInfix  
        ifTrue: [binaryCount := binaryCount + 1]  
        ifFalse: [method isKeyword  
            ifTrue: [keywordMethods add: method]  
            ifFalse: [unaryCount := unaryCount + 1]]].  
Transcript clear; show: 'Total number of methods: ', methods size printString; cr; cr;  
    show: 'Number of unaryCount methods: ', unaryCount printString; cr;  
    show: 'Number of binaryCount methods: ', binaryCount printString; cr;  
    show: 'Number of keywordCount methods: ', keywordMethods size printString.  
  
"Create an array to hold counts of keywordCount messages with the same number of arguments.  
Find its size as the maximum number of arguments over all keywordMethods elements."  
keywordCounts := Array new:  
    (keywordMethods inject: 0 into: [:max :method | method numArgs max: max]).  
keywordCounts atAllPut: 0.  
"Calculate the counts and store them in the array."  
keywordMethods do:  
    [:method| | size | size := method numArgs.  
        keywordCounts at: size put: (keywordCounts at: size) + 1].  
  
"Output results."  
Transcript cr; cr; show: 'Number of arguments/Number of methods'; cr.  
1 to: keywordCounts size do:  
    [:number| Transcript show: number printString, ' '; tab;  
        show: (keywordCounts at: number) printString; cr]
```

⁴ To find more about organization, read Appendix 1.

Note that we calculated the maximum number of arguments over keyword selectors using the inject:into: message.

Main lessons learned:

- String is an abstract sequenceable collection whose elements are character codes.
- Concrete strings are instances of String subclasses, ByteString by default. Selection of the concrete class is transparent and the programmer does not need to pay attention to it unless a larger character set is needed. In that case, a TwoByteString may be used or a new subclass defined.
- The most useful protocols of String include messages to search for substrings and to replace substrings, copy, and compare strings.
- Symbol is a subclass of String that does not allow duplication.
- The main uses of symbols are for method selectors and as widget IDs in GUI specifications.
- Binary messages are also called infix messages.

Exercises

1. Why didn't we use self instead of newString in replace:with:?
2. Does our replace:with: method work in all subclasses of CharacterArray?
3. Write a method to remove all occurrence of a substring from a given string.
4. List all collections that understand the , (comma) concatenation message.
5. Explain the definition of = in class String.
6. Study the definition of changeFrom:to:with: and show how to use it to insert 'xyz' behind the first occurrence of letter \$a in an existing string, and how to append a substring at the end.
7. One of the frequently used String methods is withCRs. Read its definition and find how it is used. Use withCRs to display a dialog request window whose prompt has 'Enter your first and last names.' on the first line, and 'Capitalize the first letter of each name' as the second line. Write the message with and without using withCRs and comment on the difference between the two styles.
8. Modify the example from the Symbol section to gather statistics on *all* classes in the library. The program should print the total number of unary, binary, and keyword messages in the library, and the selectors of all methods with the largest number of arguments. (Hint: To enumerate over all classes, use message allClassesDo: from SystemDictionary.)
9. Repeat the previous exercise for class methods. (Hint: Class methods are instance methods of the class's class and to ask an object what is its class, send it the class message. See Appendix 3 for more.)
10. Executing 'aSymbol' asSymbol returns #aSymbol. What is ('Symbol1 ' , 'Symbol2') asSymbol?
11. What is the difference between asString, displayString, and printString?
12. How many different characters can be implemented with ByteString? How many with TwoByteString?

8.8 Text - its nature and use

The purpose of Text objects is to gather all information necessary to display characters on a display medium such as the computer screen or a printer. Although you can display a String without converting it to Text because it automatically converts itself to a text object when asked to display itself, using a Text is necessary if you want to display the string with a non-default font or color, underlined, large or small, and so on.

Text has two instance variables called string and runs. Variable string holds the characters to be displayed (a String), and runs is an instance of RunArray which holds ranges of indices and encoded information about their emphasis. As an example, runs might say that the first characters use default style, the next 15 are underlined, and so on. Conversion from styles to actual fonts is performed with a TextAttributes object which indicates, for each style, the rendering parameters to be used. The concept of TextAttributes is not simple and we will not deal with it in this book.

As an example of a Text object, if we specify that string 'abcdefgh' should use emphasis #bold for elements 3 to 6, in other words should be displayed as 'abc**defgh**', the internal variables are

```
runs = RunArray (nil nil #bold #bold #bold #bold nil nil)
string = 'abcdefgh'
```

where nil means default emphasis, in other words default boldness, font, color, and size. Style #bold is a code name for a certain font converted to an actual graphical representation when the character is displayed.

The protocol of Text contains *accessing*, *emphasis*, *displaying*, *comparison*, *copying*, and other protocols and most of them are *character oriented*. As an example, *accessing* method at: anIndex returns the character at location index but not its emphasis.

Truly sophisticated display of text requires the understanding of TextAttributes and other display characteristics. As we already mentioned, these are detail beyond our scope and we will restrict ourselves to explaining how to create Text objects with simple built-in emphases and how to control text color.

The basic message to control emphasis is *emphasizeFrom: start to: stop with: emphasis* where start and stop are the indices of the endpoints of the substring to be emphasized, and emphasis is a Symbol representing emphasis - an association specifying color, or an array containing emphasis and possibly color information. Standard emphases symbols include #bold, #italic, #large, #normal, #serif, #small, #strikeout, and #underline. As the simplest example,

```
'Bold text' asText emphasizeFrom: 1 to: 4 with: #bold
```

changes the emphasis of the word 'Bold' to #bold. Note that we had to convert the string to text before we could send the emphasis message.

When you want to change the color of a string, you must define it as an Association object consisting of the symbol #color and a value specifying the desired color expressed as a ColorValue object. We will have more to say about associations later but for now, use them as in the following example:

```
'Bold text' asText emphasizeFrom: 1 to: 4 with: #color -> ColorValue red
```

The binary message -> creates an Association object with the specified key (here #color) and value (here ColorValue red). An important point about specifying emphasis is that every message that defines new emphasis overrides the emphasis assigned previously. As an example,

```
| text |
text := 'Bold text' asText.
text emphasizeFrom: 1 to: 4 with: #bold.
text emphasizeFrom: 1 to: 2 with: #italics
```

ends up overriding the #bold emphasis for the first two characters with #italics. If you want to assign multiple emphasis to a character, you must specify it as an array, as in

```
| text |
text := 'Bold italicized text' asText.
text emphasizeFrom: 1 to: 15 with: #( #bold #italics).
text emphasizeFrom: 5 to: 15 with: #bold
```

or

```
| emphases text |
text := 'Bold italicized red text' asText.
emphases := Array with: #bold with: #italics with: #color -> ColorValue red.
text emphasizeFrom: 1 to: 19 with: emphases
```

As a matter of style, using specific numbers for the start and end index as we did is not usually a good idea because your indices may be wrong and must be recalculated if you change the string.

Example 1: Experiment with emphasis and labels

Label widgets in user interfaces use text and this example is an experiment with labels, their change at execution time, and emphasis.

Problem: Design an application with the user interface in Figure 8.12. When the user clicks *New label*, a series of prompts requests a string, an emphasis (using multiple choice dialog), and the starting and end index of a substring to which the selected emphasis should be applied. The emphasis selection dialog is repeated until the user clicks *Cancel*. The program then displays the text with the specified emphases in place of the original text 'Label'.

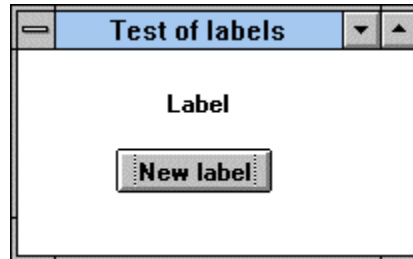


Figure 8.12. Desired user interface for Text example.

Solution: This very simple application requires only an application model class. After painting the user interface, assign #label as the ID to the label widget so that it can be accessed and modified at run time, and install the user interface on an application model. The only method that needs to be written is the *Action* method for the button:

newLabel

"Get text and emphases and display the label."

|text |

text := (Dialog request: 'Enter new text' initialAnswer: '') asText.

"Get emphases with ranges and apply them."

[| emphasis noSelection |

emphasis := self getEmphasis.

(noSelection := emphasis == #noChoice)

ifFalse: [text := self emphasizeText: text with: emphasis].

noSelection] whileFalse.

"Now send the new label to the builder for immediate display."

(self builder componentAt: #label) widget label: (Label with: text)

The two auxiliary methods used in the definition are

getEmphasis

"Obtain emphasis through dialog."

Dialog choose: 'Select emphasis for ', text

fromList: #('bold' 'italic' 'underlined')

values: #(#bold #italic #underlined)

lines: 3

cancel: [#noChoice

emphasizeText: text with: emphasis

"Apply specified emphasis to the text."

[| start end |

start := (Dialog request: 'Start index (' , text , ')' initialAnswer: '1') asNumber.

end := (Dialog request: 'End index (' , text , ')' initialAnswer: text size printString) asNumber.

^text copy "We cannot change the argument so we must work on the copy."

emphasizeFrom: start

to: end

with: emphasis

Printing text

There are two ways to print text. The more primitive one is to use the `hardcopy` message, the more sophisticated and more flexible approach uses class `Document`. We will now describe both.

Method `hardcopy` is defined in class `ComposedText` whose instances hold information about the display of text, indentation, spatial properties such as the box in which the text is displayed, and so on. Text objects themselves don't understand `hardcopy` and to display them with `hardcopy`, we must first convert them into `ComposedText`. Altogether, to display text, execute

```
text asComposedText hardcopy
```

The limitation of `hardcopy` is that it prints the whole text in the default style and ignores emphasis. To print text with its emphases, use the `Document` class.

Class `Document` has all the functionality to create documents containing text and graphics on PostScript printers. Its message `toPrinter` sends an instance of `Document` to class `DocumentRenderer` which takes care of the details of transmitting the document to the printer via class `Printer`. Both `Document` and `Printer` contain several interesting examples illustrating their functionality. In the following, we will limit ourselves to showing how to print a `Text` object with its emphasis, and how to print a window displayed on the screen (not relevant to `Text` but useful).

The general procedure for creating a text document and printing it is as follows:

1. Create a new `Document` object using the `new` message.
2. Use messages `addParagraph` and `addString: text` to define the contents of the document.
3. Close the document with the `close` message.
4. Send the document to the printer using `toPrinter`.

Example 2: Printing a Text object.

Problem: Create and print a document containing two consecutive paragraphs. The first paragraph consists of the text

Comment for class ***Document***:

which is followed by an empty line and the class comment for `Document`. The comment is printed using italics.

Solution: Following the general algorithm for creating and printing a document, the code is as follows:

```
| document string text start stop |
"Construct a Text object for the first line of document."
string := 'Comment for class Document:'.
start := string findString: 'Document' startingAt: 1.
stop := start + 'Document' size - 1.
text := string asText emphasizeFrom: start to: stop with: (#bold #italic).
"Now go into a block that creates a Document and prints it, waiting until the process has been completed."
Cursor wait "Display a special cursor while the following block is executing."
  showWhile:
    [document := Document new. "Create a Document object."
     document startParagraph. "Do the first paragraph."
     document addString: text.
     document startParagraph. "Do the second paragraph with class comment."
     document addString: (Document comment asText emphasizeAllWith: #italic).
     "Close Document and send it to the default printer."
     document close.
     document toPrinter]
```

Expression `Cursor wait showWhile: aBlock` displays the 'wait' cursor while until the block executes, in our case until the printing process gets under way.

Example 3: Printing a graphics object

The example method `doTest2` in class `Document` lets the user select a rectangle on the screen and prints it on the printer preceded with a short text. Its definition is as follows:

doTest2

"Make a simple PostScript file... a screen dump scaled in a device independent way."

| document |

Cursor wait

showWhile:

[document := Document new.

document startParagraph.

document addItalicHeader: 'Screen dump:' pixelSize: 24.

document addImage: Image fromUser.

document close.

document toPrinter]

The general procedure is the same as in the previous example but a new message called `addImage:` displays the graphics. The image itself is obtained from the user who selects it using crosshair cursors. Note that this method is based on class `Image`.

Example 4: Extend Example 1 in Appendix 1 to print the window.

Problem: Write a method to print the window from Example 1 in Appendix 1 on the printer and add a new button to the window to do this.

Solution: We have seen how to print an `Image` using `Document` and the only remaining problem is how to make an `Image` out of a window. First, we need to know how to get hold of the window object. Since the window that we want to print is active when the user clicks the button, we will need to find the currently active window. Realizing that a `Window` object is a collection of widgets, labels, measurements, and other things - and not an `Image` - we must then find how to specify the rectangle occupied on the screen by the window, and how to convert it to an image.

We will start by determining how to identify a rectangle on the screen. We note that our previous example obtains the `Image` by sending `fromUser` to class `Image`. The definition of `fromUser` is as follows:

fromUser

"Answer an instance with the same contents as a user-specified rectangle of the default screen."

^Screen default contentsFromUser

As we already know, `Screen default` provides access to the screen, and `contentsFromUser` returns the `Image` drawn by the user on the screen. Its definition is

contentsFromUser

```
"Answer an Image with the contents of a user-specified area on my screen."  
^self completeContentsOfArea: Rectangle fromUser
```

This shows that if we have a window rectangle (call it windowRectangle) we can obtain its rectangle by

Screen default completeContentsOfArea: windowRectangle

The next step is to determine how to find the active window and its screen coordinates. Since the window is the active window of our application, we can get it by sending window to the builder. When we try this and *inspect* the window, we find that the returned object is an instance of ApplicationWindow. Finally, ApplicationWindow's protocol includes message getDisplayBox which returns the window's rectangle. With this last piece of information, we can now write the complete solution of our problem:

printActiveWindow

```
"Print currently active window on printer."  
| document window windowRectangle |  
"Find window rectangle."  
windowRectangle := self builder window getDisplayBox.  
window := Screen default completeContentsOfArea: windowRectangle  
"Create document with window and print it."  
Cursor wait showWhile: "Display special cursor while the following block is evaluating."  
[document := Document new.  
 document addImage: window.  
 document close.  
 document toPrinter]
```

Add a *Print* button to the original interface in the example, add this method to the application model's class as the *Action* of the *Print* button, and test it.

Main lessons learned:

- Class Text adds emphasis to strings, allowing their display in different fonts, sizes, and colors.
- Text objects consist of a string and a run array describing character fonts and other rendering parameters.
- Text objects are usually obtained by converting a string with asText.
- For simple output of text to the printer, convert text to ComposedText and send hardcopy. This message ignores emphasis.
- For sophisticated output, possibly including graphics, use class Document.
- The builder of an application window provides access to its window.
- Windows know about properties such as screen coordinates of their bounding rectangle.

Exercises

1. Add button *Color* to Example 1. Clicking *Color* should open another multiple choice dialog listing all available colors. (Hint: See the class protocol of ColorValue.)
2. Define Text message allItalic to italicize the receiver.
3. Explain the operation of message asText in class String.
4. We simplified the nature of RunArray somewhat. Study its class definition and write a more accurate description. Explain why the description displayed by the inspector does not exactly correspond to the definition. Explain why Text uses RunArray rather than Array.
5. Write a description of class Cursor.
6. Class Screen and its superclasses provide access to some very interesting behaviors and information. Write their description.
7. Implement Example 4.

8. The method `printActiveWindow` does not contain any specific reference to the application example and can be used with any application based on the `UIBuilder`. Move it to a more general class where it could be used by any application, modify it as necessary, and test it.
9. Add a *print* command to the `<window>` menu so that the `printActiveWindow` method can be executed on any window. (Hint: Find currently active window using `Window currentWindow`.)

8.9 List widgets

In this section, we will introduce the list widget, both for its own sake and as an illustration of the use of collections. List widgets are available in two varieties - those that allow only one selection at a time, and those that allow multiple selections (Figure 8.13)

Single selection lists are much more common and they are used, for example, in the System Browser. Their purpose is similar to that of menus - making a selection - and their advantage over menus is that they allow the user to see the selections at all times - menus must be popped up or dropped down. Menus have the advantage that they don't take up space in the window. *Multiple selection lists* are used, for example, in the Definer in the UI Painter

In addition to the similarity of the purpose of lists and menus, lists are also related to check boxes and radio buttons: Single selection lists allow one selection at a time, just as radio buttons, except that in a set of radio buttons one is normally always selected which is not required in lists. Multiple lists allow any number of selections, just like check boxes. One would use radio buttons or check boxes when the number of options is small and fixed, lists would be better when the list can change or when the number of choices is large. We will now cover both types of lists starting with single selection lists.

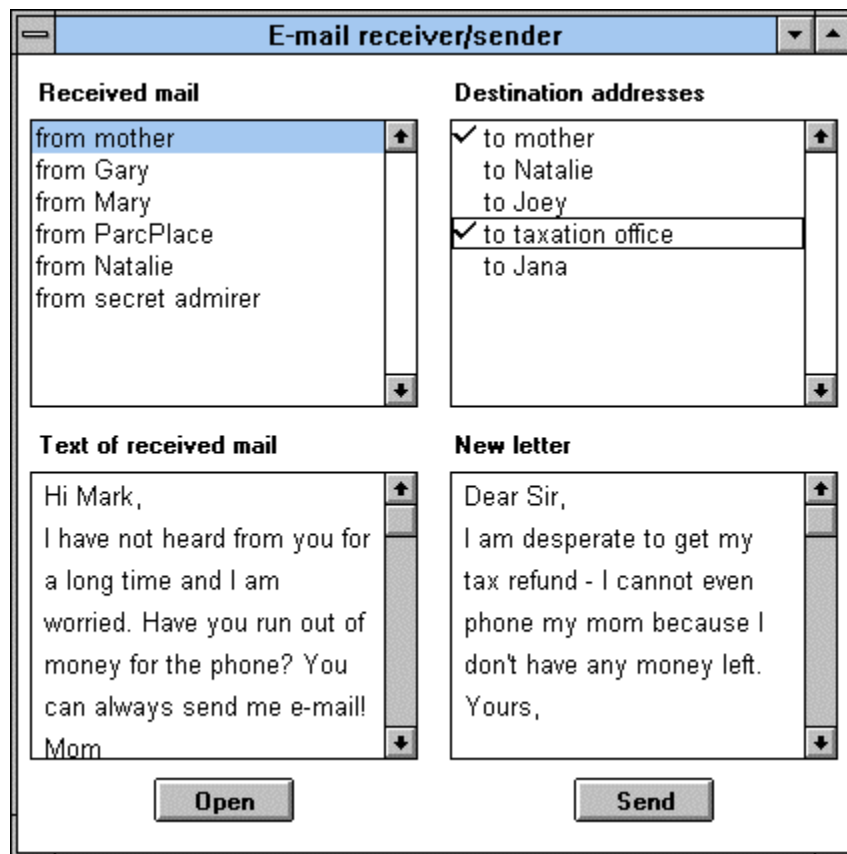


Figure 8.13. Single selection list (left) and multiple selection list (right).

Single selection lists

The basic behaviors of list widgets are: create a list widget with labels, add a new label, remove an existing label, sense change of selection, get current selection, and access labels.

The model of a single selection list (the object on which the list widget is dependent) is normally an instance of `SelectionInList`. This class has two instance variables called `listHolder` and `selectionIndexHolder`, both instances of `ValueHolder`. Variable `listHolder` holds the list of objects displayed by the widget (a `List` object), and `selectionIndexHolder` holds the index of the currently selected item or 0 if no item is selected. To be able to use single selection lists you need to know that the *Aspect* of the list widget is a `SelectionInList` and that this `SelectionInList` can be created either with message `new` as in

```
SelectionInList new
```

if the list is not available in a sequenceable, or using `with:` as in

```
SelectionInList with: aSequenceableCollection
```

where the argument contains the items to be displayed.

To display some information in the list when the window opens, initialize the list holder in the `initialize` method. To obtain the value of the currently selected item, send `selection` (returns currently selected object or nil) or `selectionIndex` (returns index of selection or 0) to the list holder. To change the selection programmatically, send `selection: aValue` or `selectionIndex: anInteger` to the list holder. To obtain the collection of items displayed in the widget, send `list` to the list holder. And to change the displayed list, send `list: aSequenceableCollection` with any sequenceable collection containing the objects for its argument. The elements of the collection should be strings. As an example,

```
listHolder list: #('line 1' 'line 2' 'line3')
```

will erase the current labels and replace them with 'line 1' 'line 2' 'line3'.

To ensure that the list notifies the application model when the selection changes, send `onChangeSend:to:` to the `selectionIndexHolder` during initialization, as in

```
classList selectionIndexHolder onChangeSend: #changedClass to: self
```

Example 1: Basic list behaviors

Problem: Implement an application with the interface in Figure 8.14 such that the window opens with labels 'label 1', 'label 2', 'label 3'. The *Add* button will display a prompt asking the user for a new label and add it to the list, the *Delete* button removes the currently selected label, and the *Print* button displays all labels in the Transcript. Every change of selection displays a notification in the Transcript. The labels must be alphabetically sorted at all times.

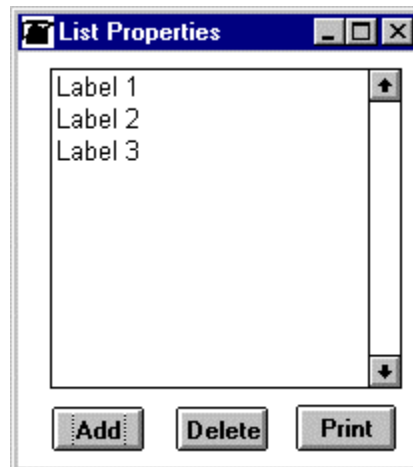


Figure 8.14. User interface for Example 1.

Solution: Paint and install the window and the widgets, and name the *Aspect* of the list (we used the name #labels). The *Action* methods will be called add, delete, and print. Since we want the window to open with three labels, we must initialize the SelectionInList and we will do this in the initialization method. The specification requires that every change in list selection trigger a message to the Transcript; we will register this interest in selection changes via the onChangeSend:to: message. The whole definition is as follows:

initialize

```
"Initialize labels in list and register interest in change in list selection."  
labels := SelectionInList with: #('label 1' 'label 2' 'label 3').  
labels selectionIndexHolder onChangeSend: #printChange to: self
```

If you now open the application, it will look exactly like Figure 8.14 but the buttons will not work and any attempt to select a label in the list will open an exception window because we have not defined printChange specified as the change message. This method, which is executed whenever the selection changes, simply asks the list for the current selection and sends appropriate output messages to the Transcript:

printChange

```
"List selection has changed, print appropriate message to Transcript."  
labels selection isNil  
  ifTrue: [Transcript cr; show: 'no selection']  
  ifFalse: [Transcript cr; show: 'selection is ', labels selection]
```

The delete method first checks whether we have a selection, and if we do, it deletes the selected label from the list. To do this, we must ask the list for its current labels (message labels listHolder value), remove the current selection from it, and assign this modified list as the new list using the list: message. It is essential to use list: because if we did not, the notification of change would not propagate to the widget and the binding between the Aspect and the list could be destroyed. The whole definition is as follows:

delete

```
"Check if there is a selection. If so, delete it from the list."  
labels selection isNil  
  ifFalse: [| newList |  
    Transcript cr; show: 'deleted ', labels selection.  
    newList := (labels listHolder value) remove: labels selection; yourself.  
    labels list: newList]
```

Note the use of yourself after remove: - it returns the modified receiver list. This is necessary because remove: modifies its receiver but returns the argument. Without yourself, the value of newList will be the string containing the deleted label, the widget will treat it as a sequenceable collection of objects, and display its individual characters as strings - instead of displaying the labels. Forgetting yourself is a very common mistake.

We used some unnecessary parentheses and an extra temporary variable to make the code more readable. Unfortunately, the method does not work because the remove: message is illegal for arrays which is the collection that we specified as underlying the list. To correct this, we convert the array to an ordered collection, changing initialization as follows:

initialize

```
"Initialize labels in list and register interest in change in list selection."  
labels := SelectionInList with: #('label 1' 'label 2' 'label 3') asOrderedCollection.  
labels selectionIndexHolder onChangeSend: #printChange to: self
```

Deletion now works and we can proceed to the add method. This method requests a new label from the user and continues much like delete except that it adds a new element rather than removing it:

add

"Request a new label. If one is supplied, add it to the list holder."

```
| string |  
    string := Dialog request: 'Enter new item' initialAnswer: ''.  
    string isEmpty  
        ifFalse: [Transcript cr; show: 'added ' , string.  
                  labels list: ((labels listHolder value) add: string; yourself)]
```

Finally the method to print all label when the print button is clicked. This method simply obtains the list from the list holder and displays its elements one after another in the Transcript:

print

"Display all labels in the Transcript."

```
Transcript cr.  
labels listHolder value do: [:item | Transcript show: item; cr]
```

Example 2: Class definition browser

Problem: Write an application with the user interface in Figure 8.15. The single selection list on the top lists all classes in the library and when the user selects one, its definition appears in the text view at the bottom.

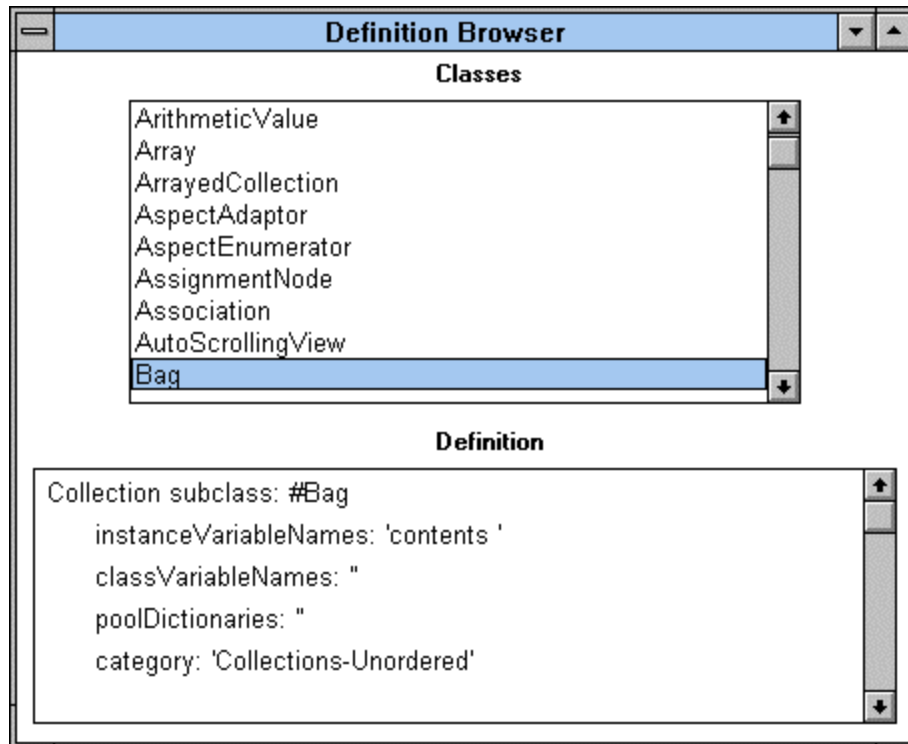


Figure 8.15. Definition browser.

Solution: As we know, class information is stored in the Smalltalk dictionary, an instance of SystemDictionary. Among its class methods is one called classNames which returns an ordered collection of symbols, the names of all classes in the library. Once we have a class, Smalltalk at: nameSymbol as in

Smalltalk at: #Array

returns the class corresponding to the symbol.

The next question is how to get the when we have the class. One way to find out is to look at the definition of `Browser` because the browser knows how to display a definition. Another possibility is to examine `Class` and other classes in its category because we know that all class information comes from this group. After a bit of searching (alternatively, you could read Appendix 3), we find that class `ClassDescription` has method definition which returns the definition of the class. Since `ClassDescription` is a superclass of `Class`, we can send the message directly to the class. As an example,

(Smalltalk at: #Array) definition

returns the definition of class `Array` in the form of a `String`.

We are now ready to implement our task. We paint the interface, *Install* it, and specify and *Define* the properties. The methods that we need include initialization (the window should open with the class list and changes in the selection list must cause changes in the text view), and the change method which responds to a new list selection and displays the definition of the selected class in the text view.

initialize

"Define the list holder for the list widget and register interest in selection changes."

```
classList := SelectionInList with: Smalltalk classNames.  
classList selectionIndexHolder onChangeSend: #changedClass to: self
```

where method `changedClass` responds to a changed class selection in the list, and changes the text displayed in the text view (*Aspect* definition). The same effect can be achieved by specifying *Notification* properties in the Properties of the List widget in the UI Painter. The definition of the message is

changedClass

"Class selection has changed. Blank the text view or display the definition of the selected class."

```
classList selection isNil  
ifTrue: [definition value: "] "Nothing selected in the list view - blank the text editor."  
ifFalse: [definition value: (Smalltalk at: classList selection asSymbol) definition]
```

Multiple selection lists

Single and multiple selection lists are essentially the same and the differences between them are as follows:

- To paint a multiple selection list, paint the List widget and click the *Multiple Selections* check box on the *Details* page of the *Properties* tool.
- The model of a multiple selection list is an instance of `MultiSelectionInList`.
- The methods accessing selections are called `selections` and `selectionIndexes`, and `selections:` and `selectionIndexes:` respectively. Message `selections` returns an ordered collection containing all selected objects, message `selectionIndexes` returns a value holder containing a set of indices. The argument of the corresponding `selections:` and `selectionIndexes:` messages must also be a collection and a set respectively.

Example 3: Function calculator

Problem: Write an application to generate and display tables of mathematical functions selected by the user from a list (Figure 8.16). If at least one function is selected, clicking *Print* prints the values of all selected functions for arguments 1 to 10 in steps of 1 in the Transcript using the following format:

arg	cos	ln	log
1	0.540302	0.0	0.0
2	-0.416147	0.693147	0.30103
3	-0.989993	1.09861	0.477121
4	-0.653644	1.38629	0.60206
5	0.283662	1.60944	0.69897
6	0.96017	1.79176	0.778151
7	0.753902	1.94591	0.845098

8	-0.1455	2.07944	0.90309
9	-0.911132	1.9722	0.954242
10	-0.839071	2.30259	1.0

When no function is selected, clicking *Print* opens a warning with appropriate text.

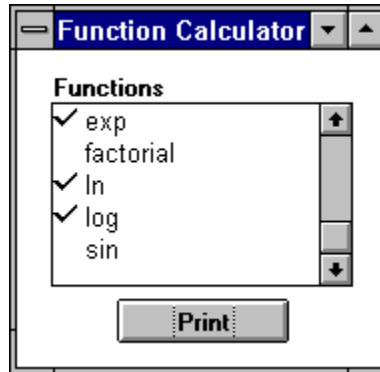


Figure 8.16. Selection window for Example 2.

Solution: The initialization method initializes the list holder variable (list *Aspect* called functions) with the names of the functions:

initialize

"Create list holder."

functions := MultiSelectionInList with: #('cos' 'exp' 'factorial' 'ln' 'log' 'sin').

When the user clicks *Print*, we enumerate over the whole interval, and for each value of the argument and for each selected function we let the argument execute the message:

print

"Display selected functions in Transcript."

functions selectionIndexes isEmpty ifTrue: [^Dialog warn: 'You must select at least one function.'].
"We have at least one selection. Clear Transcript and display heading."

Transcript clear; show: 'arg'; tab.

functions selections do: [:selection | Transcript show: selection asString; tab].

Transcript cr; cr.

"Calculate value for each argument and function and print it in Transcript."

```
1 to: 10 do: [:arg |
    Transcript show: arg printString; tab.
    functions selections do:
        [:function | Transcript show: (arg perform: function) printString; tab].
    Transcript cr]
```

Our examples show why message `onChangeSend:to:` is frequently used with single selection lists but rarely used with multiple selection lists. The reason is that it does not make sense to force the program to respond automatically to every new selection in a multiple selection list - presumably, the user will make several selections and then indicate that he or she is finished. With multiple selection lists, action is thus usually triggered by some other event such as clicking an action button.

With this example, we conclude our coverage of list widgets. They are among the most useful widgets and we will see many more examples of their use later.

Main lessons learned:

- List widgets display a list of labels and allow the user to make selections.
- Single selection lists allow one selection at a time, multiple selection lists allow any number of

selections at a time.

- The underlying data models of selection lists (their list holders) are instances of `SelectionInList` and `MultiSelectionInList`. Both hold the list of displayed items - an instance of `List` - and a value holder on the current selection or selections.

Exercises

1. Write tutorial describing how to create a list widget.
2. Extend the single selection list example into a simple browser that displays class names, definition of the selected class, and an alphabetized list of the class's instance methods.
3. In the previous exercise, add a pair of radio buttons labeled *instance* and *class* to allow the user to choose between class and instance methods.
4. Extend the previous exercise to display also the definition of the selected method.
5. Modify the multi-selection list example by adding function `cubed`.
6. Extend Example 3 to allow the user to enter the start and end values of the argument and the step.
7. In Example 3, we used variables functions and messages to hold essentially the same arrays. Remove one of them and modify the solution appropriately.
8. If you register interest in selection change, a new list selection sends a change message. Is the message sent also when the contents of the list changes?

Conclusion

In the first part of this chapter, we concluded our coverage of sequenceable (indexed) collections with ordered and sorted collections, lists, strings, text, and symbols.

Ordered collections can grow without limits and shrink (their elements can be removed) and distinguish between size and capacity. They are created with a default or specified capacity and when all slots are filled, new slots are automatically added. Growing an ordered collection is time-consuming and should be avoided by creating the ordered collection with a capacity at least equal to the maximum anticipated need. The memory space that may be wasted is generally negligible.

Ordered collections are used mainly when the number of elements is not known beforehand, or when the number of elements changes during the lifetime of the collection. They are usually accessed at the end or at the start. An important use of ordered collections is in the implementation of dependency. The major dependency protocols are defined in `Object` but the nature of the collection of dependents is redefined in `Model` which should be used as the superclass of all model objects under ordinary circumstances.

Sorted collections are ordered collections that sort their elements on the basis of a test defined by a sort block. The sort block compares two arbitrary elements of the collection and returns `true` if the first elements precedes the second element, and `false` otherwise. The built-in default sort block uses `<=` comparison but can be redefined via a class or instance message. New elements are automatically inserted into the proper place according to the sort block. Every pair of elements in the collection must be able to execute the sort block.

Class `List` combines the accessing mechanism of ordered collections with sorting of sorted collections, and adds dependency handling. Unlike sorted collections, lists do not sort themselves automatically but must be explicitly asked to do so. They sort themselves by sending a message to `SequenceableCollectionSorter`. Lists are used mainly to hold elements of multi-item GUI widgets.

Class `String` is an abstract sequenceable collection whose elements are character codes. Concrete strings are instances of `String` subclasses, normally class `ByteString`, the default string implementer. The concrete implementation of strings as `ByteString` objects is transparent and the programmer does not need to pay attention to it unless a larger character set is needed. In that case, a `TwoByteString` may be used. The most common protocols of `String` include methods for finding and replacing substrings, copying, and comparison. Much of this protocol is inherited from the abstract class `CharacterArray`.

Class `Text` adds emphasis to strings, allowing their display in different fonts, styles, and colors.

Class **Symbol** is a subclass of **String** that does not allow duplication, making its processing, in particular comparison, more efficient. The main uses of symbols are for method selectors and as names of widgets in GUI specifications.

In the second part of this chapter, we introduced two new types of selection widgets. Single-selection lists allow only one selection at a time, multiple-selection lists allow any number of selections at a time. Their basis is **MultiSelectionInList** for single-selection widgets, and **SelectionInList** for multi-selection lists. They hold a list of items and the current selection or selections. The list component of the selection in list object is an instance of **List**.

We recommend that you now read Appendix 2 which covers additional widgets.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Association, *ByteSymbol*, *CharacterArray*, *ComposedText*, *Cursor*, *Document*, *DocumentRenderer*, **List**, **MultiSelectionInList**, **OrderedCollection**, *Printer*, *RunArray*, *SequenceableCollectionSorter*, *Screen*, **SelectionInList**, **SortedCollection**, **String**, **Symbol**, **Text**, *TwoByteString*.

Widgets introduced in this chapter

Multiple selection list, *Single selection list*.

Terms introduced in this chapter

association - object consisting of a key - value pair, instance of **Association**

emphasis - specification of font, style, or color to be used when displaying a string

multiple-selection list - list widget allowing any number of selections

ordered collection - used either to refer to any collection whose elements are stored in a fixed externally accessible order, or in the more restrictive sense of an instance of class **OrderedCollection**

single-selection list - list widget allowing at most one selection at a time

sort block - a block defining how to compare two elements when sorting a collection

sorted collection - an ordered collection whose elements are ordered according to a comparison function called the sort block

symbol - a string with unique instantiation; used mainly for method selectors

text - string with emphasis prescribing how the string should display itself

Chapter 9 - Sets, bags, and dictionaries

Overview

This chapter presents the remaining frequently used collections - sets, bags, and dictionaries. All of them are unordered which means that they don't provide access to their elements in a user-defined order.

Sets are unordered collections that don't allow duplication so if you add an element to a set several times, the result is the same as if you added the element once. And if you delete an element, it is gone, no matter how many times it has been previously added. Sets are important and frequently used.

Bags are unordered collections that keep track of the number of occurrences of their elements. The Smalltalk library does not use bags and although this does not mean that they are useless, applications of bags are rare.

Dictionaries are sets of special kinds of objects called associations which are key - value pairs. For the purpose of preventing duplication, dictionaries compare association keys. Because the structure of dictionaries is substantially different from that of other collections, their protocols include a number of specialized methods. Dictionaries are used very often.

As a by-product of the discussion of sets, we will deal with the subject of copying objects. We introduce the concepts of shallow and deep copy, and point out the dangers of copying.

9.1 Sets

Sets are an important and frequently used kind of collection. Their capacity automatically grows when needed, and their elements are unordered which means that although they are internally stored as an indexed collection, there is no accessing method that allows accessing sets by index. Most often, sets are accessed by enumeration. The most important part of Set hierarchy is shown below with classes covered in this chapter in bold face.

```
Object ()
  Collection ()
    Set ('tally')
      Dictionary ()
        IdentityDictionary ('valueArray')
        PoolDictionary ('definingClass')
        SystemDictionary ('organization')
      IdentitySet ()
```

Sets eliminate duplication. This means that adding an element that is equal to an element already in the set has no effect. Since a set contains only one copy of each element, removing an element has the same effect no matter how many times the element has been added to the set before - a removed element is gone.

Creating sets

Sets are usually created with `new` and `new:`. Message `new` creates a new empty set with room for two elements:

```
new
  ^self new: 2      "Capacity 2, size 0."
```

Although sets automatically grow when their capacity is filled, the best way to create them is to try to estimate the maximum required capacity beforehand and use `new:` to create the set with a capacity about *50% larger* for better performance of element accessing.

Another creation message that is occasionally used to create sets is `with:` and its multi-keyword variants as in

Set with: 'a string' "Returns the one element set Set ('a string')"

The withAll: aCollection creation message can be used to create a set containing the elements of aCollection. The following code fragment uses this approach to create a collection of all elements of array that are divisible by 6 without duplication:

```
| array divisible |  
array := #(13 13 12 12 24 24 56 56 54 54).  
divisible := Set withAll: (array select: [:number | (number rem: 6) = 0])
```

It is instructive to see the definition of withAll:

withAll: aCollection

```
| newCollection |  
newCollection := self new: aCollection size * 3 // 2.  
newCollection addAll: aCollection.  
^newCollection
```

The method first creates a new set (self is class Set) whose size is 50% bigger than the size of the collection, adds all the elements of the original collection one after another, and returns the result - the new set. The new set thus shares its elements with the original collection and if either the set or the original modifies one of them, the other collection is affected too.

Sets can also be created with asSet which is understood by all other collections, and this method is often used to eliminate duplication. With asSet, our previous code fragment could be rewritten as follows, giving exactly the same result:

```
| array divisible |  
array := #(13 13 12 12 24 24 56 56 54 54).  
divisible := (array select: [:number | (number rem: 6) = 0]) asSet
```

Note that if our only goal was to eliminate duplication in the original values, we could proceed as follows:

```
| array |  
array := #(13 13 12 12 24 24 56 56 54 54).  
array := (array select: [:number | (number rem: 6) = 0]) asSet asArray
```

Adding, testing, and removing elements

Set messages for *adding* new elements include the usual add: anObject and addAll: aCollection. Message add: adds a new element without duplication, addAll: adds all elements of aCollection individually, again without duplication. To illustrate the difference between add: and addAll:, compare the results of executing the following two expressions:

```
Set new addAll: #(1 2 3 4); yourself.      "Returns Set (1 2 3 4) - a set with four elements."  
Set new add: #(1 2 3 4); yourself.      "Returns Set (1 2 3 4) - a set with a single element."
```

Note that we had to use yourself to obtain the resulting set because both add: and addAll: return their argument rather than the modified receiver. This behavior is, of course, identical to the behavior of add: and addAll: in all other collections.

To test whether an object is a member of the set use includes: anElement or contains: aBlock. Both messages return true or false. As an example,

```
| array set |  
array := #(13 13 12 12 24 24 56 56 54 54).  
set := (array select: [:number | (number rem: 6) = 0]) asSet.      "Returns Set (12 24 54)."
```

```
set includes: 12.                "Returns true."  
set includes: [:element| (element rem: 6) ~= 0] "Returns false."
```

Removing methods include `remove: anElement` and `remove: anElement ifAbsent: aBlock`. Both return the argument, just like `add:` and `addAll:`

```
| set |  
set := Set new addAll: #(1 2 3 4); yourself. "Returns Set (1 2 3 4)."  
set remove: 1. "Returns 1."  
set "Returns Set (2 3 4)."
```

Another message that removes set elements is the binary message `-` (minus) which calculates *set difference*. The difference of two sets is the set of those elements that are in the receiver set but not in the argument set. In other words, the difference is a set obtained by removing from the receiver those elements that are in the argument. As an example,

```
 #(1 2 3 4 5) asSet - #(1 3 5) asSet
```

returns Set (2 4).

After this overview, we will now look at several short examples.

Example 1: Is there a difference between `asSet` and `withAll:`?

Problem: `asSet` and `withAll:` seem to have the same effect. Is there any difference between them?

Solution: There are two definitions of `asSet`, one in `Collection` and one in `Bag`. The definition in `Collection` is the one that most collections inherit and its listing is as follows:

asSet

```
"Answer a new instance of Set whose elements are the unique elements of the receiver."  
| aSet |  
aSet := Set withAll: self.  
^aSet
```

This shows that `asSet` and `withAll:` have exactly the same effect and that `addAll: aCollection` is more direct.

Example 2: How does a Set eliminate duplication when adding a new element?

Solution: The definition of `add:` in `Set` is as follows:

add: newObject

```
"Include newObject as one of the receiver's elements. Answer newObject."  
| index |  
newObject == nil ifTrue: [^newObject]. "Ignore nil objects."  
index := self findElementOrNil: newObject. "Find index where newObject should be."  
(self basicAt: index) == nil "If newObject is not at this location, add it."  
ifTrue: [self atNewIndex: index put: newObject].  
^newObject
```

The definition reveals that sets are stored as indexed collections even though the `Set` protocol makes the index invisible. The definition is based on `findElementOrNil:` which is defined as follows:

findElementOrNil: anObject

```
"Answer the index of anObject, if present, or the index of a nil entry where anObject would be placed."  
| index length probe pass |  
length := self basicSize.  
pass := 1.  
index := self initialIndexFor: anObject hash boundedBy: length.  
[(probe := self basicAt: index) == nil or: [probe = anObject]]  
whileFalse: [(index := index + 1) > length
```

```

        ifTrue: [index := 1.
                pass := pass + 1.
                pass > 2 ifTrue: [^self grow findElementOrNil: anObject]]].
^index

```

Method `findElementOrNil`: tries to find the object in the collection, and if it does not find it, it returns the index where it should be stored. The principle of the search is as follows (Figure 9.1):

1. Ask the object for its hash value and scales it to the size of the set. This is the index at which the object would *ideally* be stored.
2. If the corresponding slot is empty or if it already contains `anObject`, we are finished.
3. If the slot is not empty but does not contain `anObject`, `anObject` could be stored further down. The method checks the next index, and the next, and so on, until the an empty slot or a slot containing `anObject` is found, or until the method reaches the last index. If the method succeeds, we are finished.
4. If the method reaches the last index without finding an empty slot or `anObject`, it starts pass 2 from `index = 1`. Search continues until the condition succeeds (it then exits and returns `index`) or until it fails by reaching the last available index.
5. If search in Step 4 failed, the collection is full and does not contain `anObject`. The method thus grows the collection and sends `findElementOrNil`: again. This time, the message will succeed because we have just created empty slots.

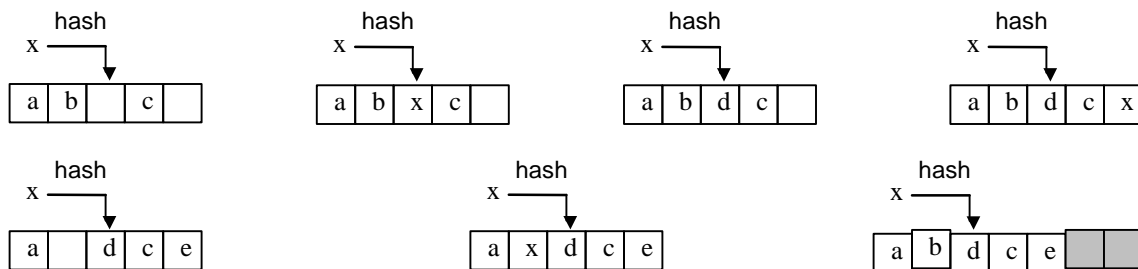


Figure 9.1. Situations that can occur while adding an element to a set. Top (left to right): Slot at hash index is empty, slot contains the argument, slot is occupied by another element but another slot further down is available, slot is occupied by another element but the element is stored further down. Bottom (left to right): No success in the first pass but available slot is found on second pass, element is found on second pass, two passes fail because collection is full and does not contain element - enlarge capacity (gray squares) and try again, this time finding an empty slot.

Our analysis of `findElementOrNil`: explains the major the role of hash - it calculates an number characterizing an object. In this case, the number is used as the first candidate for an index to store the object. The same principle is used to find an object as follows:

includes: anObject

"Answer whether `anObject` is one of the receiver's elements."

```

^self basicAt: (self findElementOrNil: anObject)) notNil

```

and hashing is also used to remove an element from the set. If the hash method is well designed and if there are enough available slots, the scaled hash values calculated for different objects will be spread out and the number of clashes between different objects producing the same hash value and competing for the same slot will be minimal. An element inserted into a set will then be accessed very fast and this is a major advantage of sets.

Since all objects must respond to the hash message, hash is defined in class `Object`. Since equality and hashing are closely related – two objects that are equal (message `=`) must have the same hash value -

hash should be redefined in all classes that redefine =. For efficiency of accessing in sets, two objects that are not equal should have a different hash value. However, this is not required and often not guaranteed.

The main use for hashing is fast access of set elements and its calculation must therefore be simple and fast. The principle of the hash method usually is that if only one component of a multi-component object is used to determine equality, its hash value is used for the hash value of the object. As an example, if class `Person` has instance variables `firstName` and `lastName` and if we compare `Person` objects by their last name only

= aPerson

`^lastName = aPerson lastName`

the definition of hash in class `Person` should be

hash

`^lastName hash`

If equality depends on more components, their hash values should be combined to get the hash function of the object and the usual way of doing this is to use exclusive or. As an example, consider a `LibraryEntry` class with instance variables `author`, `title`, `publisher`, and `year`. If we define equality on the basis of `author` and `title`

= anEntry

`^author = anEntry author and: [title = anEntry title]`

we will probably define hash in `LibraryEntry` as follows:

hash

`^(author hash) bitXor: (title hash)`

With this definition, two entries that are equal will also have the same hash value, and if the hash methods of `author` and `title` are good, the hash method of `LibraryEntry` will also behave well.

As a final note on hashing, the principle of converting an object to a representative integer is the principle of hash tables, a very efficient technique for storing performance is better when the capacity of the set is a prime number. More detailed coverage of hashing is beyond the scope of this book.

Example 3: Set union.

Problem: Set difference (the `-` message) is only one of several mathematical set operations. The other common operations are set union, intersection, and symmetric difference but these operations are not defined in the `Set` class. Define a method for *set union* as a binary message `+` using the definition that the union of two sets is the set containing all elements that are in at least one of the two sets.

Solution: The obvious solution of this problem is to take the first set and add all the elements of the second set to it as in

+ aCollection

"Calculate union of receiver and aCollection. Return the argument as in all add and remove messages."

`^self addAll: aSet`

which could be used as in

```
|set1 set2 set3|
set1 := #(1 3 5) asSet.
set2 := #(1 2 3 4) asSet.
set3 := set1 + set2           "Returns Set (1 2 3 4 5)."
```

etc.

Although this solution works, it changes the receiver `set1` and this is undesirable. To calculate the union without changing the receiver, we can use a *copy* of the receiver rather than the receiver itself as in

```
+ aSet
"Calculate the union of the receiver and the argument."
| temp |
temp := self copy. "Make a copy so that we don't change the receiver."
temp addAll: aSet.
^temp
```

Note that the argument may be any collection, not just a `Set`.

Example 4: Finding all text fonts available on the current platform

Problem: An application such as a word processor needs to know which text fonts are available on the current platform. Determine how to find their names.

Solution: Hoping that there are messages with the word `Font` in them, we opened the *implementors of* .. browser on `*Font` messages, and found one called `availableFonts`. It turned out to be an instance message in class `FontPolicy`.

To be able to use the method, we need an instance of `FontPolicy`. After some further searching we found a `FontPolicy` used as a component of class `GraphicsDevice` responsible for display devices. `FontPolicy` has an instance variable called `defaultFontPolicy` which provides access to `defaultFonts`. The next question is how to get an instance of `GraphicsDevice`. After looking at subclasses of `GraphicsDevice`, we found class `Screen` and we thus inspected

```
Screen default defaultFontPolicy availableFonts
```

and this indeed returned a large array of `FontDescription` objects. When we inspected one of them, we found that `FontDescription` has many components and one of them called `family` contains a string with the name of the font. To collect the names of all fonts, we thus inspected

```
Screen default defaultFontPolicy availableFonts collect: [:font| font family]
```

This returned a large array of strings and when we inspected them, we found that many of them are duplicates. The reason for this is turned out to be that there are usually several instance of the same family differing in various parameters such as size. To eliminate this duplication we converted the array of family names to a set by

```
(Screen default defaultFontPolicy availableFonts collect: [:font| font family]) asSet
```

This indeed returns the desired result.

Example 5. The 'lost object' paradox

Consider the following code fragment:

```
| rectangle |
rectangle := 10@10 corner: 100 @ 100.
Transcript clear; show: rectangle hash printString; cr.
rectangle origin: 20 @ 20.
Transcript show: rectangle hash printString
```

It will not surprise you that the two values of hash output to Transcript are different because the `rectangle` object, while maintaining its identity, changed its structure and thus its hash value. (When we say that "object `x` maintains its identity after a change" we mean that all objects that referred to `x` before the change still refer to it after the change. Just like John Doe remains John Doe when he moves from one

address to another or loses weight, the x object retains its identity even when it changes its state.) Now assume that we insert the rectangle into a set, change it, and then test the set for its presence as in

```
| rectangle set |  
...  
"some statements calculating set"  
rectangle := 10@10 corner: 100 @ 100.  
set add: rectangle.  
...  
"some more statements"  
...  
rectangle origin: 20 @ 20  
"some more statements"  
set includes: rectangle
```

The amazing result is that although rectangle is still in the set when we send includes:, the last statement *may* return false! The reason for this is that when we execute includes:, its findElementOrNil: method will recalculate hash (getting a different result than when it inserted rectangle into set) and the starting index for its search will thus be different from what it was when we inserted rectangle into set. findElementOrNil: will thus look for rectangle in a slot different from the one where it was initially put.

One of the bad things that can now happen is that the slot corresponding to the new index is still empty, the findElementOrNil: method concludes that the element is not in the set, and includes: returns false.

In essence, the reason for this abnormality is that the set element rectangle changed its hash value during its lifetime. To eliminate this possibility, the library contains another kind of set called IdentitySet which uses identityHash (and ==) instead of hash (and =) in its findElementOrNil: method. The difference between identityHash and hash is that the value of identityHash never changes during the lifetime of an object, even if the object itself changes. We leave the study of IdentitySet and identityHash as an exercise.

Main lessons learned:

- Sets (and other unordered collections) store their elements in a sequential fashion but don't allow index-based access.
- The most common set-related protocols are creation, conversion, enumeration, adding, and removing. All behave consistently with other collections.
- To access an element, sets calculate its hash value, a SmallInteger, using the hash method. The default definition of hash is in class Object.
- Because of their shared use in sets, hash and = are related and when = is redefined, hash should be redefined as well.
- The value of hash may change during an object's lifetime even though the identity of the object does not. This may have unexpected results when testing sets. Class IdentitySet eliminates this problem.

Exercises

1. Explain the purpose of instance variable tally in the definition of Set.
2. To create a collection of elements without duplication, you can create a set and add the elements to it, or create an OrderedCollection with the elements and convert it to a set. Which approach is faster?
3. Define Set methods to calculate symmetric difference (set of all elements that are in one set but not in the other), and intersection (set of all elements that are in both sets). The receiver should not be affected.
4. Is there a measurable difference in speed between withAll: and asSet? What do you conclude about the penalty of extra message sends?
5. Implement a new set class using equality instead of hashing to eliminate duplication and compare its performance with the existing implementation.
6. Method findElementOrNil: is the basis of accessing objects in a set. Explain how its implementation creates a close relationship between hash and = messages.

7. How does `remove:ifAbsent:` work? What is the major problem that it must resolve?
8. Explain why inspecting a `Set` often shows many `nil` elements.
9. The size of a set should always be considerably smaller than its capacity. Why?
10. We explained that if a set element changes its state during its lifetime, the test for its presence in the set may fail. This begs two questions:
 - a. Under what conditions will the test succeed, and when will it fail?
 - b. What can we do to prevent this from happening without using `IdentitySet`? (Hint: Consider what is the cause of the problem and examine the protocols of `Set` for a solution.)
11. Demonstrate the 'lost object' paradox in a code fragment. (Hint: Under what conditions is it more likely to occur?)
12. Experiment with efficiency of hashing as a function of the relation between set capacity and its size, and the choice of the capacity of the set as a prime number.
13. How does class `IdentitySet` differ from `Set`?
14. Examine ten definitions of `hash` in the library and check if each has a corresponding definition of `=`.
15. Explain the difference between `hash` and `identityHash` and illustrate it on several examples.
16. Explain the reason for the word `identity` in the name of `identityHash`.
17. The comment of `identityHash` contains the following sentence: When two objects are not `==`, their `identityHash` values may or may not be the same. Explain it.
18. Are there other uses of hashing in the class library beyond the one discussed in this section?

9.2 Copying objects

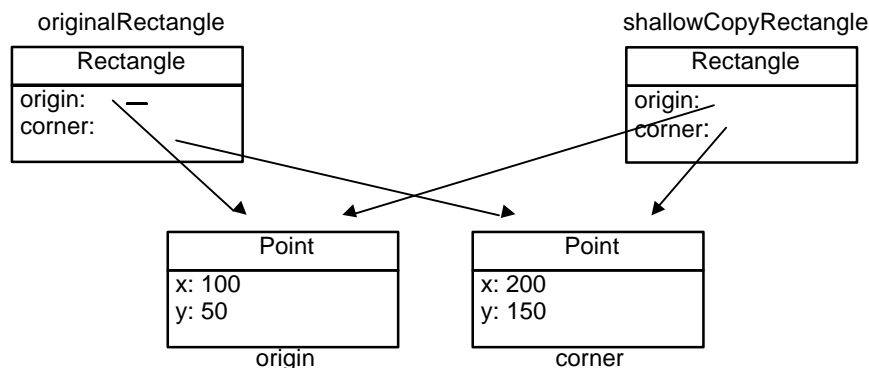
How does the `copy` method that we used in the previous section work? Does it create a new object that shares its components with the original, or is it a completely new object with new components? Or is it something in between? This is a very important question because if `copy` shares instance variables with the original then changes of the original affect the copy and vice versa, and this may not be desirable. As an example, if you wanted to make a new version of a document by making and editing a copy, changing the copy would change the original as well.

As a specific example, consider a Smalltalk `Rectangle` - an object with instance variables `origin` and `corner`, each of them an instance of `Point`. What does the following program fragment do?

```
| aRectangle aRectangleCopy |  
aRectangle := Rectangle origin: 10@10 corner: 20@20.  
aRectangleCopy := aRectangle copy
```

The two ways in which `copy` could be implemented are as follows (Figure 9.2):

- The `copy` message could create a new `Rectangle` object and make its two instance variables share its values with the instance variables of the original. This kind of copy is called a *shallow copy*.
- The message could create a new `Rectangle` and assign its instance variables copies of instance variables of the original. This kind of copy is called a *deep copy*.



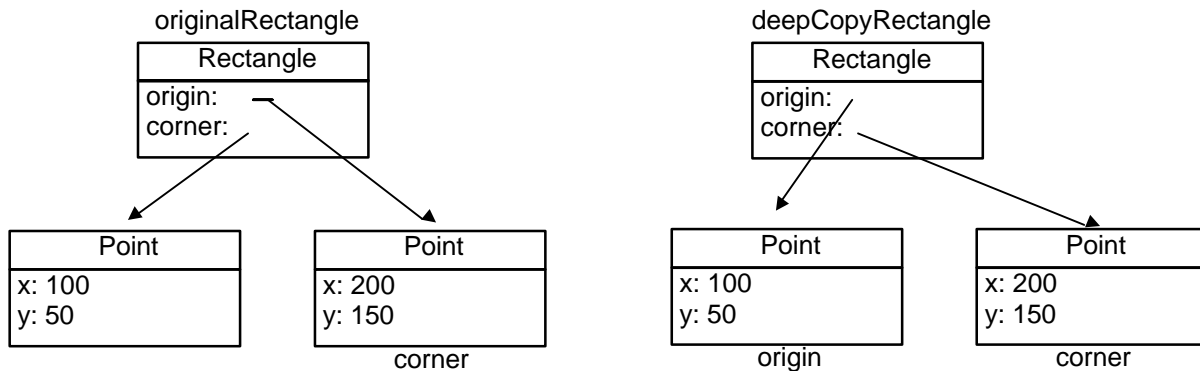


Figure 9.2. Shallow copy (top) and deep copy (bottom).

Using these new terms, we can rephrase our original question as follows: Does Smalltalk make shallow copies, deep copies, or some other kind of copies? The answer is that Smalltalk generally makes shallow copies but provides a mechanism to extend the copy operation, allowing the programmer to create a deep copy if it is needed. We will now explain how this works. The default version of copy is defined in Object as follows:

copy

```
" Answer another instance just like the receiver."
^self shallowCopy postCopy
```

The method first makes a shallow copy and then sends postCopy to the result. The default definition of postCopy does not do anything and its only purpose is to allow other classes to perform any additional work after the shallow copy has been made. In other words, postCopy is a hook.

The overwhelming majority of classes don't redefine postCopy and for most classes, a copy is thus a shallow copy. Method shallowCopy itself is almost never redefined and the only exceptions are classes that do not allow copies at all such as Symbol (each Symbol is unique) or UndefinedObject (there is only one instance - nil). As an example, the definition of shallowCopy in Symbol is

shallowCopy

```
"Answer the receiver because Symbols are unique."
^self
```

As we have already mentioned, few classes redefine postCopy and the default definition is to do nothing. The following example shows how postCopy is *redefined* in class Rectangle:

postCopy

```
super postCopy.
origin := origin copy.
corner := corner copy
```

The definition creates two new Point objects whose values are the same as those of the original and a copy of a Rectangle is thus a deep copy as in the bottom part of Figure 9.2.

Although the difference between the effects of shallow and deep copy should now be clear, we will illustrate it on a small example. The following fragment first creates a Rectangle and makes its shallow copy using shallowCopy (not the usual way of making copies) and a deep copy using copy. It then modifies the origin of the original rectangle and shows how this change affects the shallow and the deep copy:

```
| deepCopyRectangle shallowCopyRectangle originalRectangle corner |
```



```

"Create a rectangle."
originalRectangle := Rectangle origin: 10@10 corner: 20@20.
"Make shallow and deep copies."
shallowCopyRectangle := originalRectangle shallowCopy.
deepCopyRectangle := originalRectangle copy.
"Show whether instance variables are the same objects or just copies."
Transcript show: 'shallow copy corner == original corner?',
    (shallowCopyRectangle corner == originalRectangle corner) printString; cr.
Transcript show: 'deep copy corner == original corner?',
    (deepCopyRectangle corner == originalRectangle corner) printString; cr.
"Get the corner of the original and change it."
corner := originalRectangle corner.
corner x: 0; y: 0.
"Show how this affected the original, the shallow copy, and the deep copy."
Transcript cr; show: 'Rectangles: ', cr;
    tab; show: 'original rectangle: ', originalRectangle printString; cr;
    tab; show: 'shallow copy: ', shallowCopyRectangle printString; cr;
    tab; show: 'deep copy: ', deepCopyRectangle printString

```

The program produces the following output:

```

shallow copy corner == original corner? true
deep copy corner == original corner? false

```

Rectangles:

```

original rectangle: 10@10 corner: 0@0
shallow copy: 10@10 corner: 0@0
deep copy: 10@10 corner: 20@20

```

The result is as expected: Changing the corner object in the original affects the shallow copy (and the original would be similarly affected by changing the shallow copy), but not the deep copy.

Note that the side effect produced by shallow copy and illustrated by the above example occurs only if we change the *state* of a shared component but not its *identity*. If we *replace* a component rather than *change* it, we change the identity (the two objects now refer to different components) and there is no side effect (Figure 9.3). To see this, replace

```

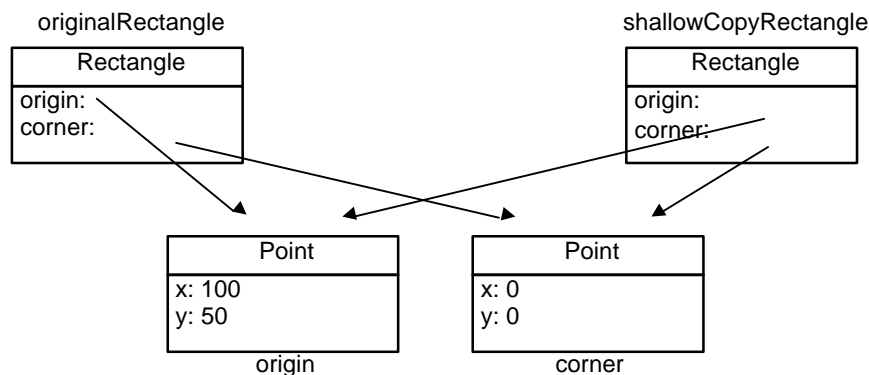
corner := originalRectangle corner.
corner x: 0; y: 0.

```

in the example above with

```
originalRectangle corner: (Point x: 0 y: 0)
```

and re-execute it. This time, the original will be different even from its shallow copy.



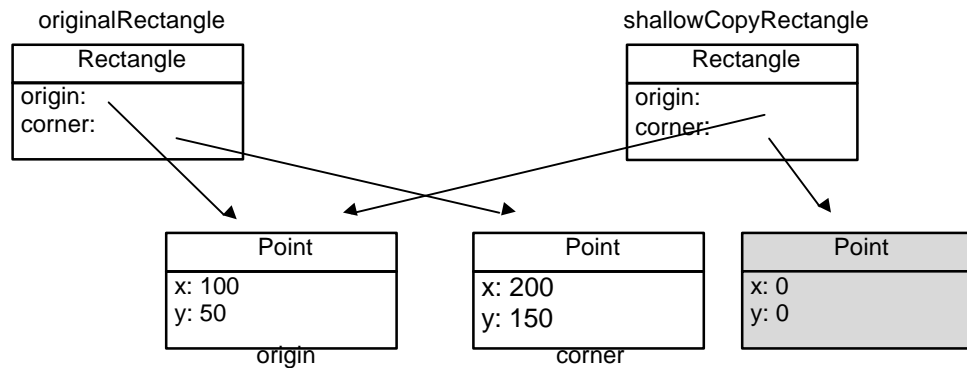


Figure 9.3. Original and shallow copy from Figure 9.2 after changing the value of corner (top) and after assigning a new Point to corner (bottom).

Up until recently, Smalltalk contained a `deepCopy` method but the method was removed because it is difficult to implement for *circular (recursive) structures* - objects, whose instance variables point to objects whose instance variables point to the original or to some other object that points to the original. A deep copy can then enter into an infinite loop, copying and recopying all objects in the component chain. Since many situations don't really require a deep copy, the method has been dropped in favor of `postCopy`.

Circular structures are not only of academic interest and occur frequently in natural situations. As an example, a Student object could have a variable called `major` whose value is an instance of class `Major` representing the field in which the student majors. `Major`, in turn, could have a variable whose value is the collection of all students with this major. By following the pointers, we thus very quickly return to the starting object (Figure 9.4) and deep copy will have to be more sophisticated than just following references and making copies of referenced objects until no more references can be found.

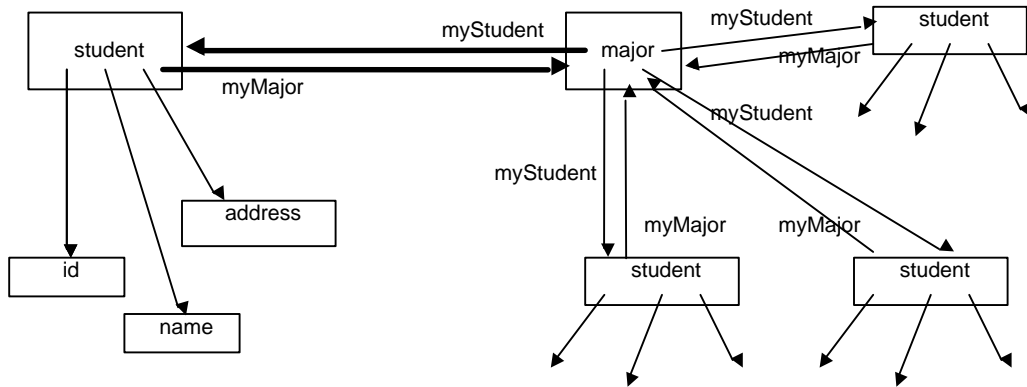


Figure 9.4. Example of a circular structure.

Main lessons learned:

- The copy operation comes in two flavors - shallow copy and deep copy.
- A shallow copy creates a new object which shares its components with the original.
- A deep copy creates a new object whose components are distinct from the corresponding components of the original but equal in value.
- Changing the state of a component of a shallow copy or its original affects both the original and the copy. This does not happen with deep copy.
- *Replacing* the component of a shallow copy or its original with a new object breaks sharing.
- VisualWorks default copy is a shallow copy followed by postCopy. The default behavior of postCopy is to do nothing but the method is redefined in several classes to obtain non-default behavior.
- The definition of shallowCopy is shared by all classes except those that don't allow a copy. These classes reimplement shallowCopy to return the receiver.

Exercises

1. Define class Student with instance variables name, a String. To test how instances of Student behave with respect to the default copy operation, create a Student object with name 'John' and its copy and observe the effect of changing the value of name in the original on the copy and vice versa. Change the name by a message such as student name at: 2 put: \$x.
2. Modify class Student to eliminate the side effect resulting from shallow copy.
3. Add class Major implementing the essence of the example in this section and making make deep copies of student-major combinations.

9.3 Bags

A bag is like a set in that it is an unordered collection. Unlike sets, however, it keeps track of the number of occurrences of each of its elements. As an example, if the same element is added to a bag five times and one copy is then removed, the bag knows that it still has four copies of the element.

Bags can be useful when we collect objects and don't care about keeping individual copies of each and the order in which they are stored, but want to know how many copies of each element we have. A typical example is, of course, putting items in a shopping bag and we will illustrate bags in this context.

Example: A better shopping minder

Modify the shopping minder program from Section 8.2 according to the following new specification:

Problem: The program allows the user (presumably a store employee) to initialize a list of items and prices via a sequence of dialogs. The program then asks the user (now the customer) to select items from a multiple choice dialog window (Figure 9.5) repeated as many times as necessary. Note that the items are displayed in alphabetical order.

When the user indicates the end of selections by clicking *Cancel*, the program prints the selected items with their counts, the total price, and all selected items costing more than ten dollars in the Transcript. The form of output is similar to that in Chapter 8. (This is still a preliminary implementation of a program with a nice user interface and file storage, but the final implementation is left as an exercise.)

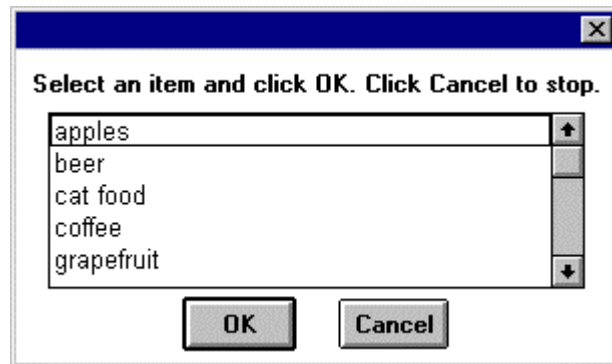


Figure 9.5. Example: Desired user interface for item selections.

Solution: Our previous implementation Item in Section 8.2 used classes ShoppingMinder. The details of ShoppingMinder will clearly change but the nature of class Item remains unchanged. We will thus restrict our attention to analyzing and modifying ShoppingMinder.

The existing ShoppingMinder stores items in an OrderedCollection called items. In the new solution, we will use this collection for the original list and add a new variable called selectedItems to hold the items selected by the customer; this variable will be used as the shopping Bag. The new ShoppingMinder thus has two instance variables called items and selectedItems and the new initialization method now has two instance variables to initialize:

initialize

```
"Initialize items to a suitably sized ordered collection."  
items := (SortedCollection new: 20) sortBlock: [:item1 :item2| item1 name < item2 name].  
selections := Bag new: 20
```

Execution of the program is the responsibility of the execute method, as before. Its essence remains valid for the new formulation except that we have now separated item entry and item selection:

execute

```
"Let store person add items, let the user select, and display the results."  
[self addItem] whileTrue.  
[self selectItem] whileTrue.  
self displayItems.  
self displayTotal.  
self displayExpensiveItems
```

It now remains to update the components of this definition. Method addItem simply asks the user to confirm that another item is to be added, requests the information, and adds it to the items variable. The old definition is

addItem

```
"Obtain an Item object from the user or an indication that no more items are needed. Add it to items."  
| name price |  
name := Dialog request: 'Enter item name' initialAnswer: " .
```

```
name isEmpty ifTrue: [^false].      "Exit and terminate loop - user indicated end of entry."  
price := (Dialog request: 'Enter item price' initialAnswer: '') asNumber.  
items add: (Item name: name price: price).  
^true
```

and although the nature of variable items has changed from `OrderedCollection` to `SortedCollection`, this does not affect the validity of the original code thanks to polymorphism - `OrderedCollection` and `SortedCollection` both respond to `add`: appropriately. This method thus remains unchanged.

Method `selectItem` is new. It opens a multiple choice dialog window with items, and allows the customer to make a selection. The definition is

selectItems

```
"Open window with available items and allow customer to make selections."  
| labels selectedItem |  
labels := (items collect: [:anItem | anItem name]).  
selectedItem := Dialog choose: 'Select an item and click OK. Click Cancel to stop.'  
                    fromList: labels  
                    values: items  
                    lines: 5  
                    cancel: [nil].  
  
selectedItem isNil  
    ifTrue: [^false]  
    ifFalse: [selectedItems add: selectedItem.  
              ^true]
```

The definition of `displayItems` is simple, we only add the display of the number of occurrences of each item:

displayItems

```
"Display alphabetically all items selected by the customer along with their prices. Use alphabetical order."  
Transcript clear.  
selections asSortedCollection do: [:item | Transcript show: item displayString; tab;  
    show: (selections occurrencesOf: item) printString; cr]
```

The code illustrates the power and flexibility of collections - the bag is easily converted into a sorted list and easily displayed alphabetically in the Transcript. The definition of `displayTotal` is almost the same as before except that we must take care of the number of occurrences of individual items:

displayTotal

```
"Add together prices of all selected items and display them in Transcript."  
Transcript cr; show: 'Total price: '; tab; show: (selectedItems inject: (Currency cents: 0)  
    into: [:total item | total + (item price * (selectedItems occurrencesOf: item))]) displayString
```

Finally, the only change in `displayExpensiveItems` is that it operates on `selectedItems` rather than `items`:

displayExpensiveItems

```
"Display a heading and all items that cost more than 10 dollars."  
Transcript cr; cr; show: 'Expensive items: '; cr.  
selectedItems do: [:item | item isExpensive ifTrue: [Transcript show: item displayString; cr]]
```

We are now finished and the last step is to test that everything works. And sure enough, when we test, the program does not do quite what we expected. The problem is that the total price seems to be incorrect as in

apple juice	price: 1 dollars 90 cents	number of units: 2
cat food	price: 24 dollars 50 cents	number of units: 1
mineral water	price: 2 dollars 50 cents	number of units: 3
Total price:	54 dollars, 60 cents	

Expensive items:
cat food price: 24 dollars 50 cents

Something is wrong with `displayTotal`, and the only thing that can be wrong is the calculation in `inject:into:`. This method is defined in `Collection` in terms of `do:` and we should thus check whether we are not misunderstanding `do:`. Checking implementors of `do:`, we find that `Bag` has its own definition

```
do: aBlock  
  contents keysAndValuesDo:      "Use block providing access to objects and their counts."  
    [:obj :count | count timesRepeat: [aBlock value: obj]]
```

which shows that for each element of the bag, the block statements are executed as many times as there are occurrences of the element and this is the cause of our problem. We must eliminate multiplication by the number of occurrences as follows:

```
displayTotal  
"Add together prices of all selected items and display them in Transcript."  
  Transcript cr; show: 'Total price: '; tab; show: (selectedItems inject: (Currency cents: 0)  
    into: [:total :item | total + item price]) displayString
```

After this change, everything works fine and the output for the same items is

```
apple juice      price: 1 dollars 90 cents  number of units: 2  
cat food price: 24 dollars 50 cents  number of units: 1  
mineral water    price: 2 dollars 50 cents  number of units: 3
```

```
Total price:      35 dollars, 80 cents
```

Expensive items:
cat food price: 24 dollars 50 cents

Main lessons learned:

- Bags are unordered collections that don't eliminate duplication.
- If the number of occurrences of an element is n , the `do:` message in class `Bag` executes the block statements with this element n times.

Exercises

1. Although bags are conceptually related to sets, this relationship is not reflected in their place in the `Collection` hierarchy. Examine the definitions of `Set` and `Bag` and write a brief comparison.
2. How does `Bag` know the number of occurrences of its elements?
3. Is `Bag`'s treatment of new elements based on equality or equivalence?
4. Write a summary of all definitions of `occurrencesOf:`.
5. Create an array containing 500 random integer numbers between 1 and 1,000 and calculate how many different integers it contains and how many copies of each. Use `Bag`.

9.4 Associations and dictionaries

Many applications involve operations on key-value pairs. Arrays and other kinds of sequenceable collections provide a specialized form of this structure if we consider consecutive integer indices to be keys, but in many cases we need a non-integer object for the key. For situations such as these, Smalltalk provides class `Dictionary`.

An obvious example of the need for dictionaries is an assembler program which translates a source program into machine instructions, replacing symbolic instruction names with their binary opcodes. This process is based on associating symbolic names with binary opcodes which could be described as

'add' -> 2r10010001 "The prefix 2r means that the code is a base 2 number. r stands for radix."
'sub' -> 2r10010010
etc.

Another possible use of dictionaries is an English-French dictionary consisting of pairs of English words and their French equivalents as in

'father' -> Set ('pere', 'papa')
'mother' -> Set ('mere', 'maman')

As yet another example, one could represent a font as a set of associations such as

'Arial' -> Dictionary ('a' -> shape of 'a', 'b' -> shape of 'b', etc.)
'New Times Roman' -> Dictionary ('a' -> shape of 'a', 'b' -> shape of 'b', etc.)

A library of fonts would then be a dictionary whose values are other dictionaries (Figure 9.6).

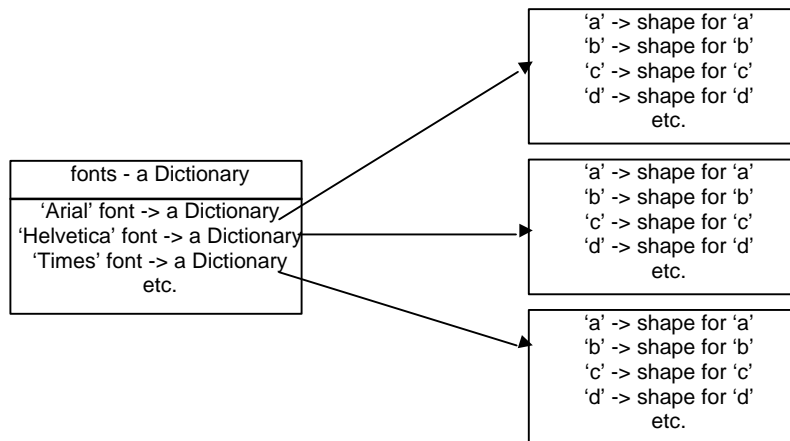


Figure 9.6. Font library as a dictionary of dictionaries.

Class Dictionary

According to its comment, Dictionary is a Set of Association objects, key->value pairs. It is a computer analogy of the conventional dictionary except that it is not ordered. Another difference is that while a conventional dictionary often include entries with the same keys, a Smalltalk dictionary does not allow two different associations with the same key because it is a Set and its comparison for equality uses = on keys. Associating a new value with a key already present in a Dictionary thus replaces the old association with the new one. As an example, if a Dictionary contains association¹

'father' -> 'dad'

and you add a new association

'father' -> 'daddy'

¹ Binary message -> which creates a new association is declared in class Object because any object may be used as the key of an Association.

the second association replaces the first one because its key is equal to the first key.

IdentityDictionary is an important subclass of Dictionary that uses equivalence instead of equality, and is internally represented as a collection of keys and a collection of values rather than as a set of associations. An IdentityDictionary is often used as a kind of a knapsack to carry around a variety of identifiable objects that could not be predicted when a class was designed and consequently could not be declared as instance variables. To put this differently, an IdentityDictionary can be used as an on-the-fly collection of instance variables. As an example, when a user interface implements drag and drop functionality (Appendix 8), the object in charge of following the mouse cursor around the screen is an instance of DragDropManager. One of the functions of this object is to carry around information about the source widget and the target widgets that might be useful in the operation. Since the nature of some of these objects is unpredictable, they can be stored in an IdentityDictionary as associations identifyingSymbol -> object. Any object that needs them may then access them by their symbolic name, almost as if they were instance variables. As another example, a software agent wandering around the Web might carry along and gather all kinds of unpredictable objects, and these objects, along with identifying symbols, could be stored in an IdentityDictionary. An ordinary Dictionary could be used as well but Symbols are unique and identity comparison is thus appropriate.

The idea of making the dictionary class a subclass of set seems strange. After all, real dictionaries are arranged alphabetically by their keys and subclassing to SortedCollection would seem more logical. The reason for this choice is efficiency: Set elements are accessed by hashing and this is a very fast operation.

After this general introduction, we will now examine the main protocols of class Dictionary and give several short examples of their use.

Creation

Dictionaries are usually created with message new which creates an uninitialized dictionary with a default capacity. If you can estimate the size, use the new: message to eliminate growing. You can also create an initialized dictionary using message withKeysAndValues: whose argument is an array of alternating keys and values as in

```
|array |  
array := #('a' 1 'b' 2 'c' 3).  
Dictionary withKeysAndValues: array      "Returns Dictionary ('a'->1 'b'->2 'c'->3)."
```

This method is rarely used.

Adding and changing elements

Message at: key put: value adds a new association or replaces an existing association that has the same (equal) key. This message is frequently used as in the following example which creates a table of natural logarithms:

```
|logarithms |  
logarithms := Dictionary new: 10.  
1 to: 10 do: [:key | logarithms at: key put: key log].  
etc.
```

Storing values in a Dictionary for easy retrieval is one of its typical uses. It is useful when we need frequent access to objects that require a long time to calculate and when we are willing to trade memory space for execution time. This approach to increasing program speed is called *caching*.

Note that at:put: returns the value of the second argument rather than the dictionary itself. This behavior is, of course, in line with behavior common to all collections.

Another way to add associations to a dictionary or to change them is to use add: anAssociation as in

```
dictionary add: key -> value
```


which has the same effect as

dictionary at: key put: value

Removing associations

To remove an association from a dictionary, use `removeKey:` or `removeKey:ifAbsent:`. The standard Collection messages `remove:` and `remove:ifAbsent:` are illegal with dictionaries because they are redefined as follows:

remove: anObject ifAbsent: exceptionBlock

```
"Provide an error notification that Dictionaries can not respond to remove: messages."  
self shouldNotImplement
```

and similarly for `remove:`.

Accessing

To obtain the value associated with a given key, use the `at: message` if you know that the key is present, or the `at:ifAbsent: message` if you are not sure. The following example allows the user to enter an English word and displays the French equivalent if the pair is in the dictionary. If the English word is not in the dictionary, it displays a 'not in the dictionary' warning:

```
| answer array dictionary key |  
"Construct a dictionary."  
array := #('mother' 'mere' 'father' 'pere' 'sister' 'soeur' 'father' 'papa').  
dictionary := Dictionary withKeysAndValues: array.  
"Prompt user for key word."  
key := Dialog request: 'Enter an English word present in the dictionary' initialAnswer: ".  
"Check whether the key is in the dictionary and construct a message."  
answer := dictionary  
    at: key  
    ifAbsent: ['not in the dictionary'].  
"Display result."  
Dialog warn: 'The French equivalent of ', key, ' is ', answer
```

Note that we put two associations with keyword 'father' in the dictionary. Check what is the result.

Two other useful accessing messages are `keys` and `values`. Message `keys` returns the *set* of all keys, message `values` returns an *ordered collection* of values as in

```
|array itemsAndPrices |  
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).  
itemsAndPrices := Dictionary withKeysAndValues: array.  
itemsAndPrices keys.    "Returns Set (#eggplant #kiwi #sugar #salt #coffee)."  
itemsAndPrices values.  "Returns OrderedCollection (20 25 15 8 35)."
```

Enumeration

Dictionaries can be enumerated with `do:` and other enumeration methods just like other collections. Remember, however, that `do:` work son *values*, not on associations as one might expect. As an example of the use of enumeration, the total price of grocery items stored in a dictionary containing item-price pairs could be calculated with `inject:into:` because its definition uses `do:` and therefore works on values:

```
| array itemsAndPrices total |  
"Create an Array with items and prices."  
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).
```

```
"Convert it to a Dictionary of symbol -> integer values."  
itemsAndPrices := Dictionary withKeysAndValues: array.  
"Calculate the total."  
total := itemsAndPrices inject: 0 into: [:subTotal :price | subTotal + price].  
total "Returns 103."
```

The reason why enumeration works on values is that Dictionary is conceptually a keyed collection, a generalization of the concepts of Array and OrderedCollection. Just as enumeration messages for Array and OrderedCollection enumerate over values rather than indices, enumeration over dictionary elements thus ignores keys. In fact, this analysis suggests that Dictionary is conceptually misplaced in the class hierarchy and that its position is dictated by implementation reasons. In other Smalltalk dialects, Dictionary is indeed located on a different branch of the Collection tree.

In addition to enumeration over values, it is often necessary to enumerate over *associations*. For this, use message keysAndValues:. As the name suggests, keysAndValues: requires a two-argument block ranging over the keys and values of individual association. The first block argument is the key and the second is the value as in the following example which uses keysAndValues: to display the pairs of item names and prices from a shopping list in the Transcript in the following format:

```
eggplant costs 20  
kiwi costs 25
```

The program is as follows:

```
[array itemsAndPrices |  
"Construct a dictionary of names and prices."  
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).  
itemsAndPrices := Dictionary withKeysAndValues: array.  
"Display the name-price pairs."  
Transcript clear.  
itemsAndPrices keysAndValuesDo: [:item :price | Transcript show: item, ' costs ', price printString; cr]
```

Since a dictionary is a set, the order in which the associations are listed is unpredictable. It is interesting to note that keysAndValuesDo: is also understood by all sequenceable collections where it treats keys as integer indices.

Testing

Message contains: aBlock is defined in Collection on the basis of detect: which, in turn, uses do:. As a consequence, Dictionary inherits contains: and applies it to values as in

```
itemsAndPrices contains: [: aValue | aValue > 10]
```

Message includes: aValue tests whether the dictionary contains an association with value aValue. To check for a *key*, use includesKey: aKey. All testing messages return true or false. As you can see, many dictionary messages access values by default.

Main lessons learned:

- A dictionary is a set whose elements are associations.
- An association is a pair of objects called key and value.
- Uniqueness in dictionaries is based on equality of keys.
- The most frequently used Dictionary messages include creation, adding, removing, accessing, and enumeration.
- Most dictionary methods work with values rather than associations.
- One of the uses of dictionaries is for caching frequently needed objects that would require considerable time to calculate. The price for improved speed is increased memory requirements.

Exercises

1. Test whether caching logarithms in a dictionary really saves time.
2. What does the add: anAssociation message return?
3. Write and test method withKeys: array1 withValues: array2 to create a new dictionary from two arrays.
4. Implement two-dimensional arrays using dictionaries. Each association's key is a row number and the value is an array of elements of the corresponding row.
5. Reimplement the two dimensional array by using an array of indices for the key and the element value as the value of the association.
6. Reimplement NDArray from Chapter 7 using a dictionary.
7. Write a description of IdentityDictionary.

9.5 Dictionary with multiple values

In many situations requiring dictionaries, the value part of each association is a collection. As an example, each English word in an English-French dictionary may have multiple French counterparts. When we add an English-French pair to such a dictionary, the message should create a new pair if the dictionary does not yet contain the English word, but if the English word already is in the dictionary the French word should be added to the existing collection of meanings. Assuming that the dictionary is stored in instance variable dictionary, this can be done as follows:

addEnglish: string1 french: string2

"Add string2 to the collection for string1 if it exists, otherwise create new string->collection association."

```
dictionary includesKey: string1
  ifTrue: [dictionary at: string1 add: string2]
  ifFalse: [dictionary at: string1 put: (Set with: string2)]
```

Checking whether a French word is among the values of such a dictionary could be done by obtaining all the values (a collection of Set objects) and checking whether any of them includes the word:

includesFrench: aString

"Check whether French word aString is in the dictionary."

```
^dictionary values contains: [:collection| collection contains: [:el| el = aString]]
```

Deletion is also possible but again not trivial.

Dealing with collection-type values is thus possible but not completely elementary. Since one of the main principles of Smalltalk is that programming should be made as simple as possible by extending the built-in facilities, it makes sense to create a new kind of dictionary whose values are collections and where manipulations of the above kind are implemented as methods. We will create such a class and call it MultiValueDictionary. The new class is a specialization of Dictionary and all methods available in Dictionary should remain executable. We will thus make Dictionary its superclass.

If we want to make our new class really useful, it should allow any type of collection as the value of its associations, not just sets as in the above example. However, it seems natural that all collections of a particular instance of MultiValueDictionary should be collections of the same kind. As an example, the methods in the example above were based on the assumption that the value is a Set and used Set with:. To make it possible to create new associations with the proper kind of value collection, each MultiValueDictionary must know what kind of collection its values should be. We will keep this information in a new instance variable called collectionClass and define the class as follows:

```
Dictionary variableSubclass: #MultiValueDictionary
  instanceVariableNames: 'collectionClass'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Chapter 9'
```

The next question is which protocols inherited from Dictionary must be redefined. The place to start is, of course, *creation*. Since our new class has a new instance variable which is essential for the operation of some of the other messages, none of the existing Dictionary creation messages is appropriate. We will thus disable all inherited creation messages (`^self shouldNotImplement`) and define new ones as follows:

newOnClass: aClass

"Create new MultiValueDictionary with default capacity on the specified class."
`^(self new: 2) collectionClass: aClass`

new: size onClass: aClass

"Create new MultiValueDictionary with specified capacity on the specified class."
`^(super new: size) collectionClass: aClass`

where `collectionClass:` is an instance accessing method of instance variable `collectionClass`. Time to test! We executed the following expression with *inspect*

`MultiValueDictionary newOnClass: OrderedCollection`

and obtained the inspector in Figure 9.7. There is obviously something wrong here - where is our `collectionClass` instance variable?

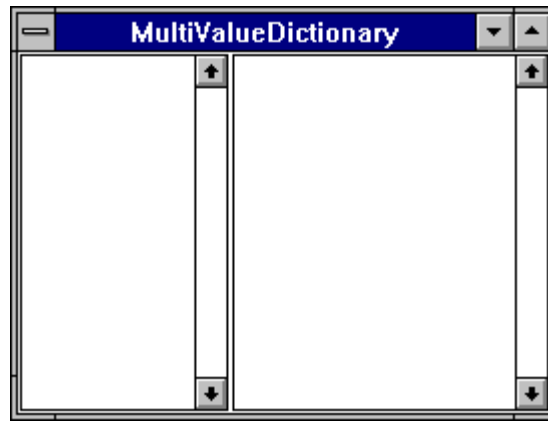


Figure 9.7. Inspector obtained for a MultiValueDictionary.

There are two possible explanations - either the inspector is 'wrong' or our definition is wrong. Since our definition is very simple and does not seem to leave room for error, we check whether Dictionary happens to have its own definition of *inspect* - and it turns out that it does:

inspect

`DictionaryInspector openOn: self`

where `DictionaryInspector` is a subclass of `ApplicationModel`. We will leave writing a new inspector for `MultiValueDictionary` to you as an exercise and restrict ourselves to redefining `printString` and using it for testing instead of the inspector. As we know, `printString` is based on `printOn:` and we will thus redefine `printOn:`. We will reuse the definition inherited from `Dictionary` to display the associations, and append the name of the `collectionClass`:

printOn: aStream

"Print dictionary associations followed by name of value class."
`super printOn: aStream.`

```
aStream nextPutAll: ', '; print: collectionClass
```

To test, we execute

```
MultiValueDictionary newOnClass: OrderedCollection
```

with *print it* and obtain

```
MultiValueDictionary (), OrderedCollection
```

as expected.

The next protocol that we must re-examine is *accessing* of keys, values, and associations. Dictionary has numerous accessing methods and we will restrict ourselves to `at: put:`, leaving the rest to you. Our new `at: put:` must first check whether an association with the specified key already exists and if it does not, it must create one with the appropriate collection as its value. If the key already exists, it will add the new value:

at: key put: anObject

"If key already exists, add anObject to its value collection, otherwise create new collection with anObject."

```
^(self includesKey: key)
  ifTrue: [(self at: key) add: anObject]
  ifFalse: [super at: key put: (collectionClass new add: anObject; yourself)]
```

Note that we used `super` to reuse the definition of `at:put:` inherited from `Dictionary`, and `yourself` to assign the collection as the value. Note also that we returned the argument to follow the general convention used by `at:put:` methods. To test the new method, we executed

```
((MultiValueDictionary newOnClass: OrderedCollection)
  at: 'Science' put: 'Computer Science';
  at: 'Science' put: 'Physics';
  at: 'Art' put: 'Music';
  at: 'Art' put: 'Drama') printString;
```

with *print it* and obtained

```
'MultiValueDictionary ("Art"->OrderedCollection ("Music" "Drama") "Science"->OrderedCollection
("Computer Science" "Physics") ), OrderedCollection'
```

Perfect! We will leave the rest to you and conclude by showing how much our new class simplifies the definition of an English-French dictionary. In the original implementation, adding a new pair of words required

addEnglish: string1 french: string2

```
^dictionary includesKey: string1
  ifTrue: [dictionary at: string1 add: string2]
  ifFalse: [dictionary at: string1 put: (Set new add: string2; yourself)]
```

Assuming that `dictionary` is a `MultiValueDictionary`, the new definition is simply

addEnglish: string1 french: string2

```
^dictionary at: string1 put: string2
```

Exercises

1. Complete the definition of `MultiValueDictionary` so that its protocols match the protocols of `Dictionary`. Make sure to redefine only those methods that must be redefined.
2. Create an inspector for `MultiValueDictionary`. (Hint: An inspector is an application.)

3. When the collection class of a MultiValueDictionary is SortedCollection, we should be able to specify its sort block. To do this, we could modify the definition of MultiValueDictionary or create a subclass of MultiValueDictionary called, for example, SortedMultiValueDictionary. Compare these two approaches and implement the one that you consider better.
4. Compare our definition of MultiValueDictionary with MultiValueDictionary defined as a subclass of Object with instance variables collection and collectionClass.
5. What requirements must a class satisfy so that it can be used as the argument of the new:onClass: message?

9.6 Example - A Two-Way Dictionary

In this section, we will design and implement a computerized dictionary for two languages such as English and French. This application, an example of the use of dictionaries, will be designed according to the following specification.

Specification

The program provides access to two complementary dictionaries using the interface in Figure 9.8. The information kept in the two dictionaries is complementary in that one dictionary can at any time be constructed from the other dictionary and any change made in one dictionary is automatically reflected in the other dictionary.

Each word in each language may have multiple equivalents in the other language. Selecting a word in the list of 'originals' in the user interface displays all its equivalents in the associated list of 'translations' and selections in the two lists of originals are independent. Control over the dictionaries is via pop up menus in the list of 'originals' according to details described below.

The application is opened by executing a Smalltalk opening message which identifies the languages to be used to label the user interface.

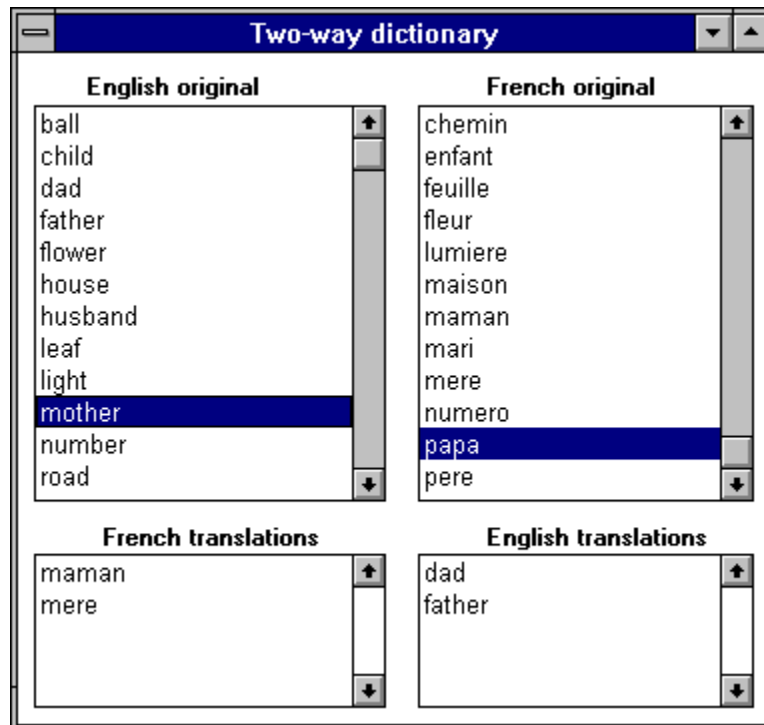


Figure 9.8. Desired interface of the two-way dictionary.

The pop up menu of both lists of original words is as in Figure 9.9. Its commands allow the user to add a new pair of words, add a new meaning to a word selected in the list, delete the selected word, and correct its spelling.



Figure 9.9. Pop up menu of the 'original' list widgets.

Solution: The problem is conceptually simple and we will implement it using an application model class, and a domain model class called `TwoWayDictionary`. We will leave the application model class as an exercise and concentrate on the domain model class.

Class `TwoWayDictionary`

Class `TwoWayDictionary` holds two complementary dictionaries², one from language 1 to language 2, the other from language 2 to language 1. Its responsibilities include instance creation, and methods implementing the menu commands in Figure 9.9.

The class will hold the names of the two languages in instance variables `lang1` and `lang2`. The two complementary one-way dictionaries will be stored in instance variables `lang1lang2` and `lang2lang1`. Instance variable `lang1lang2` will hold the language 1 → language 2 dictionary, `lang2lang1` will map language 2 to language 1. The key of each association will be a string and the value will be a sorted collection of strings (a not so accidental use for class `MultiValueDictionary` from the previous section). Figure 9.10 is an example of a possible state of the two dictionaries; note that all words appear in both dictionaries.

French - English part		English - French part	
'pere'	'dad' 'daddy' 'father'	'dad'	'papa' 'pere'
'mere'	'mother' 'mummy'	'father'	'papa' 'pere'
'papa'	'dad' 'daddy' 'father'	'daddy'	'papa' 'pere'
'maman'	'mother' 'mummy'	'mother'	'maman' 'mere'
		'mummy'	'maman' 'mere'

Figure 9.10. Typical dictionary components of a two-way dictionary for English and French.

Class `TwoWayDictionary` is a domain model, therefore not a subclass of `ApplicationModel`. Since the library does not contain any related class, it will be a subclass of `Object`. Its definition is

```
Object subclass: #TwoWayDictionary
  instanceVariableNames: 'lang1 lang2 lang1lang2 lang2lang1'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Book'
```

and its comment is

Class `TwoWayDictionary` implements two complementary dictionaries such as English-French and French-English. In each of the two dictionaries, one key may have any number of meanings (values). Entering a key-value pair into one of the two dictionaries automatically updates the other dictionary.

² Note that we are using delegation to implement a complex dictionary with simpler dictionaries.

Instance variables are

```
<lang1>      - name of the first language, a String
<lang2>      - name of the second language, a String
< lang1lang2 > - one language dictionary, a MultiValueDictionary on SortedCollection
< lang2lang1 > - the other language dictionary, a MultiValueDictionary on SortedCollection
```

In both dictionaries, keys are strings (words in one language) and values are sorted collections of strings - equivalents of the key word in the other language.

Having decided these basic features, we will now list the selectors of some of the methods and write the corresponding short methods or their descriptions. The remaining methods are left as an exercise.

Creation will be implemented by method on: lang1String and: lang2String. This class method creates a new TwoWayDictionary on languages named lang1String and lang2String. Example of use:

TwoWayDictionary on: 'English' and: 'French'

The underlying dictionaries will be two multiway dictionaries using SortedCollection as their value. The creation message is

```
on: lang1String and: lang2String
  ^ (self new) initialize; lang1: lang1String; lang2: lang2String
```

and the initialization method creates the appropriate MultiValueDictionary objects:

```
initialize
  lang1lang2 := MultiValueDictionary newOnClass: SortedCollection.
  lang2lang1 := MultiValueDictionary newOnClass: SortedCollection
```

We test our definition with the above statement using *inspect* and everything works fine.

Addition will be implemented by instance method addLang1:lang2 which adds a pair of words, one to each, without damaging the information already in the dictionary. Example of use:

```
addLang1: 'dad' lang2: 'pere'
```

The problem is simple because the MultiValueDictionary does all the hard work, and the corresponding definition is

```
addLang1: lang1String lang2: lang2String
  "Update both dictionaries."
  lang1lang2 at: lang1String put: lang2String.
  lang2lang1 at: lang2String put: lang1String.
  ^lang1String -> lang2String
```

Note that we return the argument as all add methods do. A test with

```
| dict |
dict := TwoWayDictionary on: 'English' and: 'French'.
dict addLang1: 'dad' lang2: 'papa';
  addLang1: 'dad' lang2: 'pere';
  addLang1: 'father' lang2: 'papa';
  yourself
```

confirms that everything works.

Deletion will be implemented by instance method `language1Delete: aString` (and a symmetric method `language2Delete: aString`). The method deletes the whole `language1` association with key `aString`, and propagates the change to `language2`. Since this and similar messages are sent by the pop up menu after the user has made a selection in the user interface, we can assume that `aString` is present in the set of keys of `language1`. We can thus use `removeKey:` and similar messages without worrying about the `ifAbsent:` part. The algorithm for responding to pop up command *delete* when a word in language 1 is selected is as follows:

1. Obtain all language 2 meanings of the association whose language 1 key is to be deleted.
2. Delete the `language1-language2` association.
3. Find all language 2 -> language 1 associations corresponding to the words found in Step 1 and delete the language 1 word from their value collection. (The language 1 word is guaranteed to be there.)

and the corresponding definition is as follows:

language1Delete: aString

```
"Remove language1 word aString from both dictionaries."  
| meanings |  
"Get all language2 meanings for aString."  
meanings := lang1lang2 at: aString.  
"Remove aString from all associations in language 2 containing them."  
meanings do: [:meaning] (lang2lang1 at: meaning) remove: aString.  
"Remove association with key aString from language 1. We know that aString is an existing key."  
lang1lang2 removeKey: aString
```

The definition of `language2Delete:` is similar and we leave it to you to test that both methods work.

Exercises

1. Complete the definition of class `TwoWayDictionary` and test it.
2. Create the application model of the dictionary application and test it.
3. Our popup menus are in English. Modify the design to make it possible to translate the commands to another language. The technical term for such an extension is internationalization and VisualWorks provides several tools to simplify it. These tools are not included in the basic library but you can solve the problem without them.

9.7 A Finite State Automaton

As an example of the use of dictionaries, we will now develop a class implementing a finite state automaton (FSA). A finite state automaton is a very important theoretical concept used in several areas of computer science including hardware design, compiler design, and software engineering.

The principle of an FSA can be formulated as follows (Figure 9.11): An FSA consists of a collection of *states*, each performing some *actions*. When an FSA enters a state, it executes all actions associated with this state and makes a transition to a new state depending on the outcome of these actions. Typically, one of the actions executed in each state is a response to an event such as user input. FSA operation starts in a well-defined *initial* state and continues until the FSA reaches one of its *terminal* states where it stops.

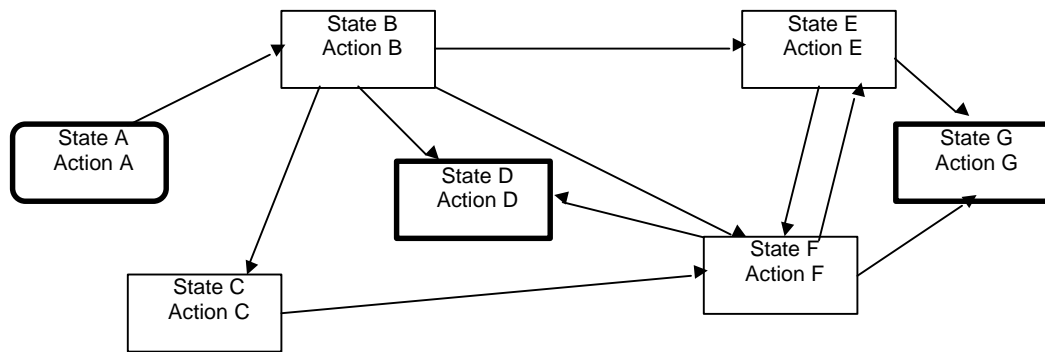


Figure 9.11. An FSA with states represented as rectangles and transitions represented as links between states. The initial state is shown as an oval, final states are thick line rectangles.

Since all FSAs work on the same principle, we can implement an FSA as a class whose instances are FSAs with specific behavior defined at the time of their creation. Each FSA must know

- all its states and their associated actions,
- transitions corresponding to action outcomes,
- the initial state,
- the terminal states,
- the current state

For practical reasons, we will also require that an FSA know what to do if the requested destination state does not exist - because the creation message did not provide a proper FSA description.

To describe state-action pairs, we will use a dictionary whose keys are state objects (for example integers or symbols) and whose values are blocks defining the actions executed in this state. Each block must return the next state and thus define a transition.

To illustrate this approach, the state behavior of the FSA depicted in Figure 9.12 can be described by a dictionary containing the following associations which ignore the failure mechanism:

key	value
1	[Transcript show: 'state 1'. ^(DialogView request: 'Enter name of next state (1 or 2 to continue, 3 or 4 to quit)') asNumber]
2	[Transcript show: 'state 2'. ^(DialogView request: 'Enter name of next state (1 to continue, 3 to quit)') asNumber]
3	[Transcript show: 'state 3 - end'. ^3]
4	[Transcript show: 'state 4 - end'. ^4]

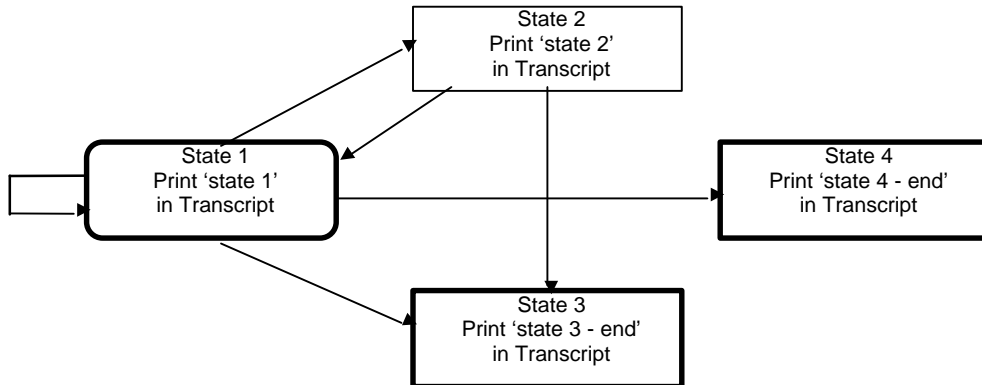


Figure 9.12. Example FSA with transitions obtained from user.

The FSA works in a loop that can be described as follows:

```

current state := initial state.
[current state := value of block corresponding to current state.
currentState is valid ifFalse: [currentState := failBlock value]]
current state is an end state] whileFalse.
self executeState      "To execute action in terminal state."
  
```

After this analysis, we can now implement class FSA. Its definition is as follows:

```

Object subclass: #FSA
  instanceVariableNames: ' currentState endStates initialState stateDictionary failBlock '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Book'
  
```

The comment is

I implement an abstract finite state automaton.

Instance variables:

stateDictionary	<Dictionary>	- keys are states, values are blocks; each block returns a next state
endStates	<Set>	- a Set of terminal states, subset of stateDictionary keys
current state	<anObject>	- current state object - a key in stateDictionary
initialState	<anObject>	- initial state - a key in stateDictionary
failBlock	<aBlockClosure>	- block executed on any attempt to make an illegal transition; must return a next state

As usual, we will start with a class method to create a fully specified FSA. The definition is

```

onDictionary: aStateDictionary initialState: anInitialObject endStates: aSet failBlock: aBlock
"Create an initialized FSA."
  
```

```
^self new
  onDictionary: aStateDictionary
  initialState: anInitialObject
  endStates: aSet
  failBlock: aBlock
```

where the instance method initializing the created FSA is simply

```
onDictionary: aDictionary initialState: anInitialState endStates: aSet failBlock: aBlock
"Initialize FSA."
  stateDictionary := aDictionary.
  initialState := anInitialState.
  currentState := anInitialState.
  endStates := aSet.
  failBlock := aBlock
```

The operation of the FSA follows our algorithms and is defined by the following method:

```
run
"Iterate until FSA enters one of the terminal states."
  currentState := initialState.
  [currentState := self executeState.
  (stateDictionary includesKey: currentState)
  ifFalse: [currentState := failBlock value].
  endStates includes: currentState] whileFalse.
  self executeState "Execute terminal state."
```

Method `executeState` evaluates the action block of the current block and returns the result, presumably the next state. The definition is as follows:

```
executeState
"Execute action and make transition."
  ^(stateDictionary at: currentState) value
```

We will now test FSA on the example in Figure 9.11. To do this, we must create the complete description of the state graph, use it to create the FSA, and start FSA operation. The operation is controlled by the user via dialogs:

```
| dictionary fsa |
"Create state dictionary."
dictionary := Dictionary new.
dictionary at: '1'
  put: [Transcript show: 'state 1'; cr.
        Dialog request: 'This is state 1.\nEnter name of next state (1 or 2 to continue, 3 or 4 to quit)' withCRs initialAnswer: '1'];
  at: '2'
  put: [Transcript show: 'state 2'; cr.
        Dialog request: 'This is state 2.\nEnter name of next state (1 or 2 to continue, 3 or 4 to quit)' withCRs initialAnswer: '2'];
  at: '3'
  put: [Transcript show: 'state 3'; cr. Dialog warn: 'This is state 3 - a terminal state'. '3'];
  at: '4'
  put: [Transcript show: 'state 4'; cr. Dialog warn: 'This is state 4 - a terminal state'. '4'].
"Create FSA."
fsa := FSA
  onDictionary: dictionary
  initialState: '1'
  endStates: #('3' '4') asSet
  failBlock: [Dialog request: 'Illegal choice; try again' initialAnswer: ''].
Transcript clear; show: 'Sequence of states: '; cr; cr.
"Run."
```

fsa run

The program runs correctly.

Exercises

1. Use FSA to simulate the JK flip-flop. The JK flip-flop is a binary storage element that stores 0 or 1 and changes state depending on its current state and two inputs called J and K. Its transitions are shown in Figure 9.13. Assume initial state 0, have user enter the two inputs.
2. Use FSA to implement a recognizer that reads and calculates non-negative integer numbers such as 13, 0, and 383888495000101 digit-by-digit. The digits are entered by the user with a Dialog request: initialAnswer: message. The machine returns the calculated value when it encounters a non-digit character.
3. The classical application of FSAs is to 'recognize' input sequences Use the FSA developed in this section to
 - a. recognize legal Smalltalk identifiers (letter followed by a sequence of letters or digits). Input is via a letter-by-letter dialog, output is to the Transcript. The output will show the entered string followed by the word 'legal' or 'illegal'. This kind of operation is performed by compilers.
 - b. unlock a door when the user enters the combination '2', '7', '9', '3', '9'.
4. Find how the Smalltalk compiler recognizes identifiers and relate this to Exercise 5.
5. A frequent programming task is reading text, processing it according to some fixed scheme (*filtering* it), and printing the result. Implement a filter defined as follows:
 - Replace every blank line preceded and followed by a line of text with a line of asterisks.
 - Replace every block of two or more blank lines between two text lines with one line of dots.
 - Leave lines with text unchanged.

As an example of the desired behavior,

Line 1.

Line 2.

Line 3.

Line 4.

should be replaced with

Line 1.

Line 2.

Line 3.

.....

Line 4.

This behavior can be described as in Figure 9.14. Implement the filter using FSA, extending our definition if necessary.

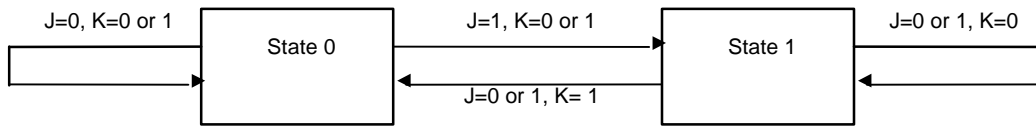


Figure 9.13. State transition table of a JK flip-flop.

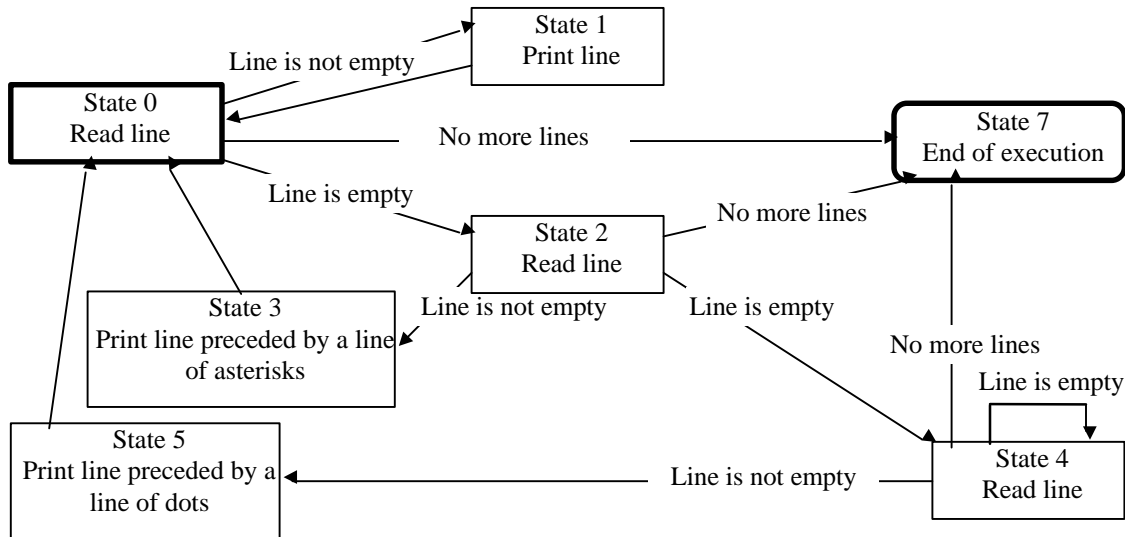


Figure 9.14. FSA for processing lines of text.

Conclusion

Sets, bags, and dictionaries are unordered collections which means is that their protocol does not allow element accessing by an index. Although the external interface is non-sequenceable, internal implementation is based on sequential storage and uses the `basicAt:` method inherited from `Object`.

Sets eliminate duplication: No matter how many times a particular object is added to a set, the set contains only one occurrence and if this occurrence is deleted no other copies of the element remain in the set. One of the main uses of sets is to eliminate duplication (often by the conversion message `asSet`). Testing whether an element is already in the set is based on hashing - calculation of an integer characterizing the object. A subclass of `Set` called `IdentitySet` uses identity hash for comparison to eliminate the possibility of changed hash values during an object's lifetime. If the hash function is well designed and the set is not too full, hashing allows very fast element access and this is another reason while sets and their subclasses are very popular.

Bags keep track of the number of occurrences of their elements both during addition and during removal. They are little used but their ability to keep the count of the number of copies of individual elements while eliminating their storage is useful.

Dictionaries are sets whose elements are associations, in other words pairs of key-value objects. Because of their special structure, Dictionary protocols contain a number of specialized messages and even the messages shared with other collections sometimes have specialized behaviors. Uniqueness of dictionary elements (the essential property of sets) is implemented by comparison of keys. A frequently used subclass of `Dictionary` called `IdentityDictionary` uses identity hashing and equivalence instead of hashing and equality for comparison. A very important subclass of `Dictionary` is `SystemDictionary` which holds information about the structure of the system and its shared objects. Its only instance is `Smalltalk`.

When using an object, it is often essential to decide whether we want to use the object itself or its copy. In many cases, we must use a copy because the original must not be affected by subsequent

operations. A copy may be shallow (new object which shares instance variables with the original), deep (completely separate from the original), or intermediate, obtained in a special way. When using a shallow copy, remember that state changes of the components of the original affect the copy and vice versa.

The difficulty with deep copy is that it is not easy to implement when the copied objects have circular structure. In VisualWorks, copy is implemented as a combination of shallow copy and a post-copy operation which makes it possible to implement any form of copying by redefining the `postCopy` method. The default behavior is equivalent to shallow copy and it is used by most classes.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Association, *Bag*, **Dictionary**, IdentityDictionary, IdentitySet, **Set**, SystemDictionary.

Terms introduced in this chapter

finite state automaton - an abstraction of a machine mechanically changing its state in response to actions associated with states

filter - a process that mechanically transforms an input (usually text) into a new form

hash - an integer characterizing an object; used, for example, to place or find an object in a set efficiently; depends on object's state and may change during its lifetime

deep copy - a copy completely disjoint from the original

identity hash - hash that does not change hash value during an object's lifetime

post copy - method executed by the copy method after a shallow copy is made; allows customization of copying mechanism

shallow copy - a copy in which the top level object is new but its components are shared with the original; the basis of the copy operation in Smalltalk

Chapter 10 - Streams, files, and BOSS

Overview

Sequenceable collections are often processed in linear order, one element after another. Although linear access can be performed with collection accessing and enumeration methods, Smalltalk library contains a group of classes called streams that simplify linear access and improve its efficiency. It is important to understand that streams are not a new kind of collection but rather a mechanism for accessing an existing collection.

Smalltalk distinguishes two kinds of streams with largely overlapping protocols - internal and external. Internal streams are used to access sequenceable collections whereas external streams are for file access.

Storing data in a file and reading it back requires two facilities: access to the contents of the file (provided by external streams) and access to the file system itself (for operations such as accessing directories and files, and for creation, naming, and deleting files and directories). Access to the file system is provided by class `Filename`. Most file operations thus require both an instance of an external stream and an instance of `Filename`.

External streams provide byte-by-byte access to file contents but no tools to store objects, thus lacking the facility that most Smalltalk programs need. Although every class knows how to convert its instances into text representing executable Smalltalk code, this facility is too inefficient for larger objects. VisualWorks thus provides a special group of classes for storing objects as binary codes. This tool is called the Binary Object Streaming Service (BOSS). Since storage and retrieval of binary objects depend on files and streaming access, the use of BOSS requires understanding of external streams and `Filename` objects.

10.1 Introduction to streams

Sequenceable collections must often be accessed one element after another with intermediate processing, as if viewed through a window that remembers which element is being viewed (Figure 10.1). Another, and historically more relevant analogy, is that a stream is like a digital magnetic tape whose recordings (collections of sound codes) are read one after another in the order in which they were recorded. This kind of access is called *streaming* and although it can be achieved with standard enumeration methods, the Smalltalk library provides a group of classes that makes streaming access easier and more efficient. A stream is thus a tool for viewing existing sequenceable collections - a collection accessor. A stream is *not* a new kind of collection.

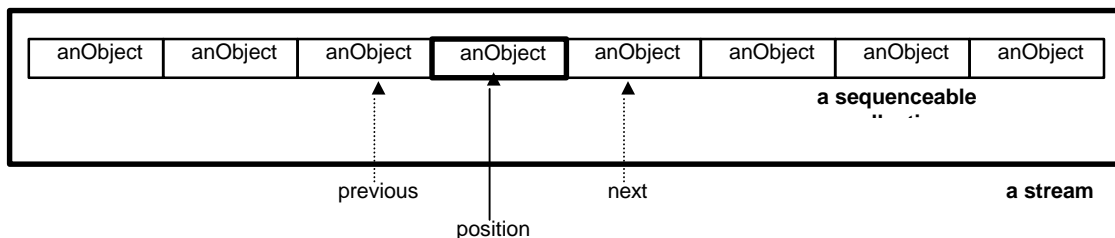


Figure 10.1. Stream is a mechanism for accessing a sequenceable collection via a positionable window.

Here are a few examples of situations that require streaming:

- Construction of text from strings extracted from a file or entered by the user. Examples include creation of reports and form letters.
- Analysis of text such as extraction of words from an article or processing of Smalltalk source code by the compiler.

- Reading and writing of files.

Execution of each of these tasks involves some or all of the following operations:

- Opening a stream on a collection.
- Selecting the starting position.
- Moving to the next or the previous element and examining its contents.
- Replacing the object at the current position with another object.
- Adding an object behind the object processed so far.
- Peeking ahead at the next element without changing the current position.
- Repositioning the stream pointer to the beginning or the end of the stream, or to any location given by an index.
- Testing whether the position pointer is at the end of the stream.
- Accessing file elements.

Since different tasks require different types of streaming access, Smalltalk streams are implemented by a group of classes, a subtree of the abstract class `Stream`. `Stream` factors out the shared properties of all streams such as having contents (the underlying collection), testing whether the end of the stream has been reached, and moving the position pointer. Some of these methods are completely defined in the abstract class `Stream` and possibly overridden at lower levels, others are defined as 'subclass responsibility'.

An example of a stream operation shared by all types of streams is enumeration. Its implementation is the same for all streams and class `Stream` thus contains its full definition:

do: aBlock

"Evaluate aBlock for each of the elements of the receiver."

[self atEnd] whileFalse: [aBlock value: self next] "Evaluate block with successive elements."

Method `next` which is the basis of the method is left as subclass responsibility.

To emphasize the close relationship between streams and collections, all classes in the `Stream` hierarchy with the exception of `Random` are defined in category `Collections - Streams` even though the `Stream` subtree in the class hierarchy is totally disjoint from the `Collection` subtree¹. The whole subtree is as follows:

Object

Stream

PeekableStream

PositionableStream

ExternalStream

BufferedExternalStream

ExternalReadStream

ExternalReadAppendStream

ExternalReadWriteStream

ExternalWriteStream

InternalStream

ReadStream

WriteStream

ReadWriteStream

TextStream

Random

¹ Class `Random` is a subclass of `Stream` only because its elements are obtained in a linear fashion. Unlike other streams, elements accessed by `Random` don't exist independently and are created when requested by message `next`.

As we have already suggested, streams can be classified according to several parameters. The first distinction used in the class hierarchy is whether the stream allows reading the next element and returning to the original position; in other words, whether it is possible to "peek" ahead without moving the cell (window) pointer. With elements generated by random number generators, this is obviously not possible since a random number generator cannot be asked to recall the random number that it generated before, and this is where Random splits from other streams in the Stream tree. Since we have already covered random numbers, the rest of this chapter deals with peekable collections only.

The fact that we can peek ahead does not imply that we can reposition the window to any place in the stream, in other words, jump from one place to another. This additional property is needed, for example, for random access of files, and its underlying mechanism is defined in class PositionableStream via its instance variable position. Its value is an integer number, an index that points to the current position of the window on the stream, an element in the underlying sequenceable collection. Most stream accessing operations first move the pointer by one position "to the right" (increment the index) and then access the corresponding element. The pointer thus always points before the element that will be accessed by the next stream accessing message. Since the index of the first element in a stream is 1, resetting a stream sets position to 0. To provide control over positioning limits, PositionableStream has two instance variables called readLimit and writeLimit. These two integers determine the current last position accessed by the stream; the first position is always the element at index 1 of the underlying collection.

Class PositionableStream is the root of two sub-trees - internal streams and external streams. *Internal streams* are used for accessing sequenceable collections residing, in principle, in the internal memory of the computer. Smalltalk uses internal streams extensively to construct messages, menu labels, arrays of coordinates of geometric objects, parsing during the compilation of Smalltalk programs, and in other operations. *External streams* are an extension of the stream concept to files. They are used to read or write elements of files stored on external media such as disks or obtained from the network.

One important difference between internal and external streams is in the *kind of objects* stored in their underlying collections (Figure 10.2). Elements of collections accessed by *internal* streams can be any objects such as integers, characters, strings, rectangles, or even other streams. *External streams*, on the other hand, are byte-oriented which means that their elements are individual bytes such as ASCII characters or binary codes with another interpretation. Byte orientation of external streams is due to the fact that files are managed by operating system functions, and operating systems access consecutive elements of files as bytes.

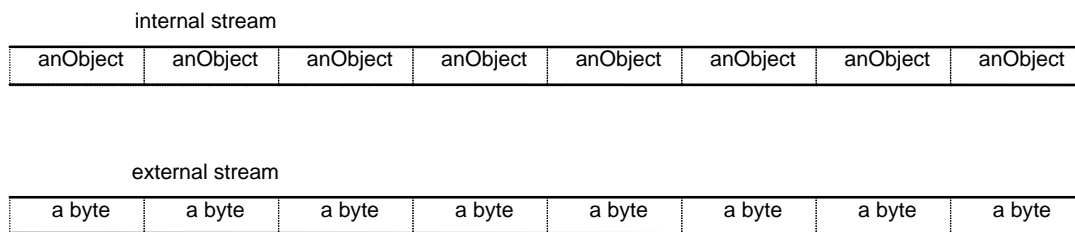


Figure 10.2. *Internal streams* may stream over collections containing any objects but *external stream* access is byte-oriented.

Another difference between internal and external streams is that their hierarchy contains an additional abstract class called BufferedExternalStream. This class implements the concept of a *buffer*, a memory area holding the working copy of a portion of a file (Figure 10.3).

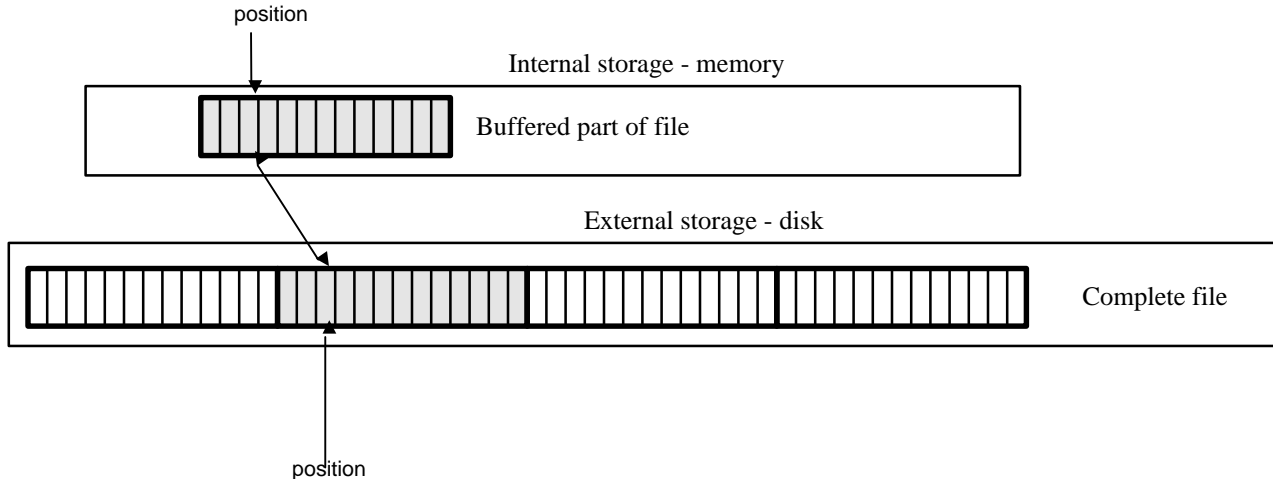


Figure 10.3. Only a part of a file is kept in the memory.

Another difference between internal and external streams is that internal streams include class `TextStream` specialized for accessing `Text` objects. External streams, on the other hand, include appendable streams that allow adding information only at the end of a file which does not have an equivalent among internal streams.

Besides the distinction between internal and external streams, we can also distinguish streams that can only be read from streams that can only be written, and streams that can be either read or written. This classification applies both to internal and external streams although there are a few differences in details.

After this brief overview, we will now present internal streams. External streams, files, and related concepts of external storage are covered in the second part of this chapter.

Main lessons learned:

- A stream is an accessor of consecutive elements of sequenceable collections.
- The two main groups of streams are internal and external streams.
- Internal streams are used to access sequenceable collections whose elements may be arbitrary objects stored in memory. External streams are used to access consecutive bytes stored in a file or on the network.
- Besides the distinction between internal and external streams, Smalltalk also distinguishes between read-only, write-only, and read-write streams.
- The class hierarchies of internal and external streams are somewhat different. The hierarchy of external streams includes class `BufferedExternalStream` which is responsible for hiding the fact that only a part of a file is present in memory at any time, internal streams include `TextStream`.

10.2 Internal streams

The Smalltalk library uses internal streams a lot but novice programmers often neglect them, probably because their functions can be implemented by operating directly on their underlying collections. Or possibly because there is such an overwhelming number of stream methods, some of them with rather obscure behaviors. This is unfortunate because stream methods considerably simplify frequently needed operations in the same way that specialized enumeration methods simplify specialized enumeration. Moreover, streams may significantly improve performance, for example as an alternative of string concatenation. And finally, most uses of streams depend on only four or five simple messages.

In the rest of this section, we will outline stream protocols, and the next section will give examples of their use. Note that although most streaming methods are shared by all stream classes, some are not:

There are methods that only work with external streams, methods that can be used with read streams but not with write streams, and so on. Most of these limitations are obvious and natural.

Creation

Internal streams are usually created with class methods `on:`, `with:`, or by messages addressed to the underlying sequenceable collections; rarely, streams are created with `on:from:to:` and `with:from:to:`. All these methods create a new stream over the specified collection and initialize the `position`, `readLimit`, and `writeLimit` variables. The details are initially a bit confusing because each method initializes these variables differently but you don't have to think about the details in most cases because the typical behavior is quite natural.

It is interesting to note that creation methods succeed even if the underlying collection is not sequenceable (for example a `Set`) but any subsequent attempt to access a stream created over such a collection will fail. Now for the details:

`aStreamClass on: aCollection`, creates a stream over `aCollection` and positions the pointer *at the start*, to `position = 0`. The initial settings of the `readLimit` and `writeLimit` depend on the kind of stream and the effect is summarized in Figure 10.4.

	position	readLimit	writeLimit
<code>ReadStream</code>	0	end of collection	end - irrelevant
<code>ReadWriteStream</code>	0	0	end of collection
<code>WriteStream</code>	0	0 - irrelevant	end of collection

Figure 10.4. Effect of `on:` on various types of internal streams.

`aStreamClass with: aCollection`, creates a stream over `aCollection` and initializes `position`, `readLimit`, and `writeLimit` to the *last* index, positioning the pointer *at the end*. To remember the difference between `with:` and `on:`, use the mnemonic that *the first letter of with: is 'at the end of the alphabet' whereas the first letter of on: is 'at the start of the alphabet'*. The effect of `with:` is summarized in Figure 10.5. with differences with respect to `on:` italicized.

	position	readLimit	writeLimit
<code>ReadStream</code>	<i>end of collection</i>	end of collection	end - irrelevant
<code>ReadWriteStream</code>	<i>end of collection</i>	<i>end of collection</i>	end of collection
<code>WriteStream</code>	<i>end of collection</i>	<i>end - irrelevant</i>	end of collection

Figure 10.5. Effect of `with:` on various types of internal streams.

The following are examples of the effect of several stream creation messages:

<code>ReadStream on: #(1 3 'abc')</code>	"Opens a read stream on array #(1 3 'abc'); position is initialized to 0."
<code>ReadStream with: #(1 3 'abc')</code>	"Opens a read stream on array #(1 3 'abc'); position is initialized to 3."
<code>WriteStream on: (String new: 16)</code>	"Opens a write stream on an empty string; position is initialized to 0."
<code>ReadStream with: ('abcd') asSet</code>	"Succeeds but any attempt to access the stream will fail."

Most stream applications use the `on:` creation message and only a few use `with:`. Creating a new stream with message `new` is illegal because it does not specify the underlying collection.

Instead of creating a stream by sending a creation message to a *stream* class, you can also create a stream by sending `readStream`, `writeStream`, or `readWriteStream` to a *sequenceable collection* as in

```
#(12 43 23 67) readStream
```

which produces the same result as

ReadStream on: #(12 43 23 67)

Accessing

This protocol includes many instance messages that return the contents of the stream (the underlying collection), reposition the pointer, or access elements of the underlying collection. The 'setting' messages (various forms of put which add one or more new elements) grow the underlying collection if necessary. Some of the messages in this protocol are:

size - returns the larger of readLimit and position. If position is larger, it increases readLimit to position.

contents - returns a *copy* of the *part* of the underlying collection from the start to the readLimit of the stream. Its definition is

contents

"Answer a copy of the receiver's collection from 1 to readLimit."

readLimit := readLimit max: position.

^collection copyFrom: 1 to: readLimit

next is used for reading the next element. It first moves the pointer to the right by one position (increments position by 1) and returns the element at this position. If the pointer is already at the end of the stream (measured with respect to readLimit or writeLimit), next returns nil and does not change the pointer.

nextPut: anObject - increments the pointer and stores anObject as the next element of the underlying collection. Returns anObject just like other adding messages. Overwrites the existing element of the collection if there was one at this position, and grows the collection if the new element is being added to a full collection. It is important to note that the stream does not work with a copy of the collection but with the collection itself.

nextPutAll: aSequenceableCollection - stores individual elements of aSequenceableCollection of size n as the next n elements of the stream. Compare this with nextPut: which would add the whole collection as a single element (Figure 10.6). The difference is similar to the difference between add: and addAll: collection messages. Another similarity between add methods in collections and nextPut in streams is that they all *return the argument* rather than the modified receiver.

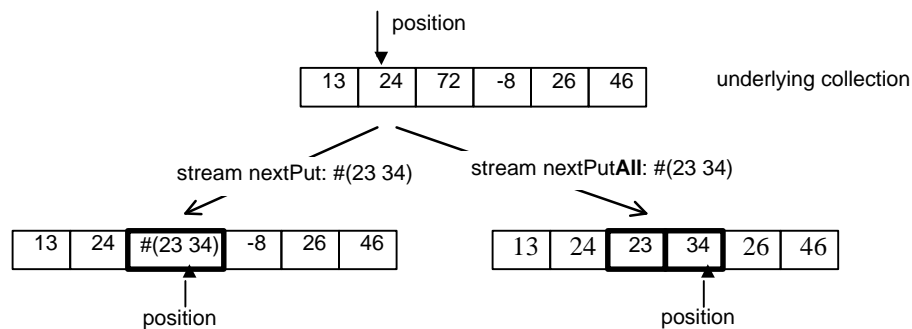


Figure 10.6. Result of `nextPut:` (left) and `nextPutAll:` (right). Note which elements are added and which elements are gone.

peek - increments position and returns the element at that position like next, but resets the pointer to its original place.

upTo: anObject - repeats sending **next** until it reaches the first occurrence of **anObject** or the **readLimit**. It returns a *collection* whose elements² are the elements retrieved by the consecutive **next** messages from the start of the iteration up to but *not including* **anObject**. The pointer is left pointing at **anObject** so that the next **next** message will return the item following **anObject**. If **anObject** is not found, the message returns a collection containing all elements from the current position up to and including the last element of the receiver stream.

through: anObject - has the same effect as **upTo:** but **anObject** *is* included in the returned stream. The final value of position is the same as for **upTo:**.

Positioning

Methods in this protocol reposition the pointer without retrieving or storing elements.

position: anInteger - changes the value of pointer to **anInteger**. This method is used mainly for reading and the value of **anInteger** is usually between 0 and **readLimit**. Remember that the element accessed by **next** will be the element at position **anInteger** + 1.

reset - resets the pointer to 0 to prepare for access to the first element. Same as **position: 0**.

setToEnd sets pointer to the last element of the stream marked by **readLimit**.

skip: anInteger - jumps over the specified number of elements without accessing them. Performs

self position: position + anInteger

In other words, method **skip:** performs *relative* repositioning with respect to the initial position, whereas **position:** is for *absolute* repositioning. As a consequence, **skip: -1** may be legal but **position: -1** never is.

skipUpTo: anObject - skips forward to **anObject** and leaves pointer pointing at it. Next access will thus be to the element following **anObject**. Returns the receiver stream on success, **nil** if it does not find **anObject**.

skipSeparators - skips a sequence of any of the following characters: space, cr, tab, line feed, null, and form feed. This and some other methods hint that internal streams are often used for character processing.

do: - uninterrupted enumeration over the underlying collection until **self atEnd** returns true. Since it uses **next** to access the consecutive elements, it *starts at the current position* rather than at the start of the collection. As a consequence, it *may not enumerate over all elements of the collection*.

Testing

Testing messages determine whether the stream is empty, what is the current position in the stream, and whether position points at the end.

atEnd - returns true if position is greater than or equal to **readLimit**. If the stream is not defined over the whole underlying collection (e.g. **on:from:to:**), **readLimit** does not refer to the last element of the collection.

² We will use 'stream elements' to refer to the elements of the underlying collection.

isEmpty - tests whether position = 0, in other words, refers to how much of the collection has been viewed. This is somewhat confusing because it is not clear what it means that a stream is empty. As an example

(ReadStream on: 'abcd') isEmpty

returns true although the underlying collection is not empty.

position - returns the current value of the pointer.

The following code fragment illustrates some of these new messages and more examples will be given later:

stream	
stream := ReadStream on: #(13 3 'abc' 'xyz' \$a \$b).	"Creates new stream over the specified array."
stream contents.	"Returns #(13 3 'abc' 'xyz' \$a \$b)."
stream position.	"Returns 0 - stream is positioned to read the first element."
stream next.	"Returns 13, the next element of the underlying collection."
stream skip: 2.	"Increments position by 2 and returns receiver stream."
stream next.	"Returns 'xyz' and increments position."
stream skip: 20.	"Opens an Exception notifier - position out of bounds."

Main lessons learned:

- The main stream protocols are creation, accessing, positioning, testing, and enumeration.
- The essential stream messages are on:, with:, next, nextPut:, nextPutAll:, and testing.
- Stream creation messages create a stream over a collection and position a pointer at the start or at the end of the underlying collection.
- A stream may be opened over a sub-range of the underlying collection.
- The values of readLimit and writeLimit represents the effective end of the stream.
- The most common accessing messages are next and nextPut:. Both first increment the pointer and then access (and possibly change) the collection.
- Positioning messages are used for random (non-linear) access.
- Details of stream messages depend on the kind of stream.

Exercises

1. What is the relationship between the position and the index in the underlying collection?
2. Examine what happens to the underlying collection when you add new elements at the end of a write stream.
3. Examine what happens to the underlying collection when you add new elements at the end of a write stream opened over its sub-range.
4. How does on:from:to: work and how does it limit the new stream's access to a part of the underlying collection?
5. Message upTo: anObject returns a subcollection of the stream's collection. What happens when you then send nextPut: to this stream?
6. What does skipSeparators return?
7. printString for streams returns only the name of the class. Redefine it to return class name followed by contents, position, and (depending on the kind of stream) the value of readLimit and writeLimit.
8. Can any other enumeration methods in addition to do: be used on streams?
9. Explain the result of each of the following lines:
(ReadStream on: 'abcdef') next; next; position: 3; next
(WriteStream on: Array new) nextPut: \$a; nextPut: 13
(ReadStream on: 'abcdef') peek; peek
(WriteStream on: String new) nextPut: \$a; nextPut: \$b; nextPut: 3

```
(WriteStream with: 'abcd') nextPutAll: 'xyz'; yourself
(WriteStream on: 'abcd') nextPutAll: 'xyz'; yourself
(WriteStream with: 'abcd') nextPutAll: 'xyz'; contents
(ReadStream with: 'abcd') position: 2; nextPutAll: 'xyz'; contents
```

10. How does contents work on writeable streams?

10.3 Examples of operations on internal streams

In this section, we will give several examples of stream behavior and demonstrate some of the most common uses of streams.

Example 1: Stream enumeration

As we already mentioned, stream implementation of `do:` operates only over the elements following the current position. Because the method does not reset the pointer when it ends, the pointer ends up pointing at the end of the stream. The method returns the receiver stream. As an example,

```
| stream |
stream := ReadStream on: 'abcdefg'.      "Creates a stream on characters; position = 0."
stream skip: 2.                          "Value of position is now 2."
stream do: [:element | Transcript show: (element printString)]
```

prints 'cdefg' and returns the read stream with position = 7.

Example 2: Using streams to edit strings - filtering

Problem: String modification is a typical use of internal streams. Write method `replace: char with: aString` to replace all occurrences of character `char` with replacement string `aString`. As an example,

'This is ~ true' `replace: $~ with: 'not'` "Should produce 'This is not true'."

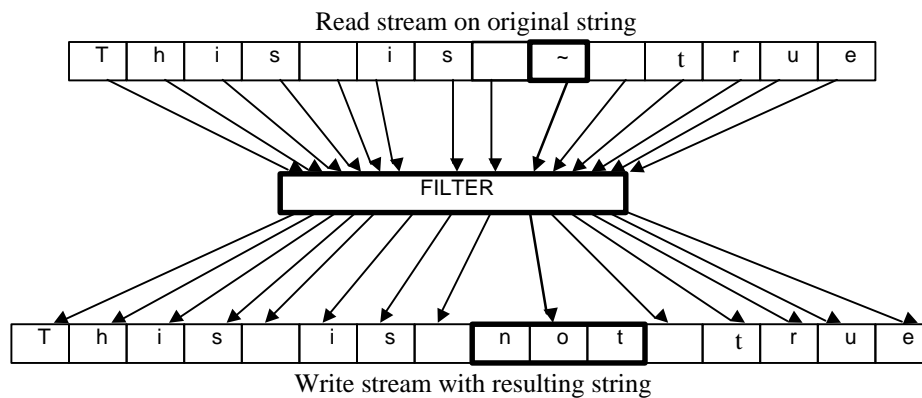


Figure 10.7. Principle of solution of Example 2.

Solution: To solve this problem (Figure 10.7), we will create a `ReadStream` on the string entered by the user, create an uninitialized `WriteStream` of suitable size for creating the output, and process the `ReadStream` one character after another, copying all characters except for `$~` into the `WriteStream`, and replacing every `$~` character with 'not'.

We will put `replace: char with: aString` in class `CharacterArray` and its definition is as follows:

replace: char with: aString

"Replace all occurrences of char with aString."

| output input |

"Open ReadStream on string."

input := ReadStream on: self.


```
"Create a WriteStream on a String."  
output := WriteStream on: (String new: self size).  
input do: [:ch | "Use the stream to build the result."  
    ch == char  
        ifFalse: ["Make replacement on match."  
            output nextPut: ch]  
        ifTrue: ["Leave other characters unchanged."output nextPutAll: aString]].  
"Returns processed string."
```

This is a typical use of internal streams - scanning a `ReadStream` and constructing a `WriteStream` a piece at a time. Note the use of `contents` to obtain the resulting string. Since the elements of the underlying `String` are characters, we use `nextPut`: to enter the unchanged *characters* but `nextPutAll`: to enter the *string* 'aString' as a sequence of characters.

Example 3: Constructing a string using a stream

Problem: As an experiment with the use of streams, write a code fragment to create a simple personalized letter from a pre-stored template. The letter is a reminder that a book borrowed from the library is overdue and it should have the following form:

May 23, 1997

Dear Ms. Jones,

I would like to remind you that the book borrowed from the Xaviera Library is now **overdue**.

Yours,

Ivan Tomek
Adjunct Librarian

The program should automatically calculate the date, the user selects one of Mr. Mrs. or Ms. from a multiple choice dialog, and the names of the borrower and the Adjunct Librarian are entered by the user. (Unfortunately, I am usually the one who gets the reminders rather than the one who issues them.)

Solution: In this program - another typical application of internal streams - we will use a `TextStream` because it can handle emphasis and inherits messages for inserting carriage returns, tabs, and other useful characters. We start by opening a `TextStream` of suitable size, construct the text from strings that are either predefined or calculated or selected by the user, and return the resulting `Text` object. The principle is simple and the code is as follows:

```
| labels letter |  
labels := #('Miss' 'Mr.' 'Mrs.' 'Ms.').  
"Create a TextStream on a String of suitable length."  
letter := TextStream on: (String new: 170).  
"Construct letter."  
letter emphasis: nil; cr; cr;  
    nextPutAll: (Date today printString); cr; cr;  
    nextPutAll: 'Dear ';  
    nextPutAll: (Dialog  
        choose: 'Which one do you want?'  
        fromList: labels  
        values: labels  
        lines: 4  
        cancel: ['']);  
    space; nextPutAll: (Dialog request: 'Enter borrower's name' initialAnswer: '');  
    nextPut: $,; cr; cr; tab;  
    nextPutAll: 'Please note that the book which you borrowed from our Library is now '; cr;  
    crtab: 3; emphasis: #(#bold #underline); nextPutAll: 'overdue';  
    emphasis: nil; cr; cr; cr; "Set and clear emphasis"
```

```
nextPutAll: 'Yours,'; cr; cr; cr;  
nextPutAll: (Dialog request: 'Enter Adjunct Librarian's name' initialAnswer: ''); cr;  
nextPutAll: 'Adjunct Librarian'
```

Test the program and print the letter – the contents of the letter `TextStream`. Use class `Document`. Note again that this example is only an illustration of the use of streams. To implement the problem of creating form letters, we would have to create one or more classes to perform the task in a more general context.

Example 4: An example of `TextStream` methods

As an example of how `TextStream` handles character oriented operations, the definition of `cr` inherited from `Stream` is

```
cr  
"Append a return character to the receiver."  
self nextPut: Character cr
```

and this is then used with the `tab` method to define `crtab` as follows:

```
crtab  
"Append a return character, followed by a single tab character, to the receiver."  
self cr; tab
```

Example 5: Skipping up to a specific character

A compiler skips over characters such as spaces and line feeds which don't have any effect on execution. This is implemented by messages such as `skipTo:`, `upTo:`, and others. We will now illustrate this principle by reading a string entered by the user and converting it into an array of strings corresponding to sections of the original terminated by `$-`. As an example, if the user enters the string

```
'This is-not-my day'
```

the program will convert it to

```
#('This is' 'not' 'my day')
```

The basis of the solution is message `upTo: anObject` which returns the collection of objects preceding the next occurrence of `anObject` or the tail of the stream; it returns an empty collection when issued at the end of the collection. The message sets the pointer to `anObject` so that next access starts just behind it. Our program again first opens a `ReadStream` on the original string, and then constructs the resulting collection by streaming.

```
|stream collection string |  
"Create OrderedCollection to hold the result – we cannot predict the eventual size."  
collection := OrderedCollection new.  
stream := ReadStream on: (Dialog request: 'Enter text using - as separator' initialAnswer: "").  
[(string := stream upTo: $-) isEmpty] whileFalse: [collection addLast: string].  
collection asArray  
"Convert because the specification required an Array."
```

Example 6: Using `with: to access the whole underlying stream`

When you create an instance of `ReadWriteStream` *on* an existing stream, its position is initialized to 0. As a consequence, a message such as

```
(ReadWriteStream on: 'A string') contents
```

returns an empty string and stream size returns 0. If you then add a new element with `nextPut:`, it will replace the first element of the original collection, and repeated use of `nextPut:` will eventually destroy all

original data. If you want to be able to access the contents of the whole underlying collection or add elements at the end, use the with: creation message as in

```
| rwStream |
(rwStream := (ReadStream with: 'A string') nextPutAll: '!!!'; yourself) contents.
rwStream nextPutAll: ' And another string!!!'.
rwStream contents "Returns 'A string!!! And another string!!!'"
```

Example 7: The use of internal streams is not limited to strings

Although internal streams are used mainly for operations on strings, they work with collections of any objects. In fact, enumeration methods such as collect: and select: defined in class SequenceableCollection are based on internal streams. As an example, method reverse which returns a copy of a collection with its elements in reverse order is defined as follows:

reverse

"Answer a new sequenceable collection with its elements in the opposite order."

```
| aStream |
aStream := WriteStream on: (self species new: self size).
self size to: 1 by: -1 do:
    [:index | aStream nextPut: (self at: index)].
^aStream contents
```

Example 8: Streams can make code more readable

Since operations on streams are actually operations on their underlying collections, what do we gain by using streams? One advantage of streams is conceptual clarity and simplicity. As an example, the following two code fragments have exactly the same effect but the second formulation is more natural, simpler and less error prone because we don't have to deal explicitly with the position pointer:

"Displaying selected elements of a collection. Implementation with collection."

```
|array position|
array := #('a' 'b' 'c' 'd' 'e').
position := 1.
Transcript show: (array at: position); cr.
position := position + 2.
Transcript show: (array at: position); cr.
position := position + 1.
Transcript show: (array at: position); cr.
etc.
```

"Displaying selected elements of a collection. Implementation with stream."

```
| array stream |
array := #('a' 'b' 'c' 'd' 'e').
stream := ReadStream on: array.
Transcript show: (stream next); cr.
stream skip: 1. "Note that we had to increment the pointer by 2 in the previous version."
Transcript show: (stream next); cr.
Transcript show: (stream next); cr.
etc.
```

Example 9: Stream operations are often more efficient

A classical example where streams improve execution speed is concatenation. The following two code fragments produce the same string but the implementation with concatenation is many times slower than the implementation with streams.

"Test of concatenation. Implementation with string concatenation."

```
Time millisecondsToRun: [
    | string |
    string := 'abcdefg'.
    1000 timesRepeat: [string := string , 'abcd']] "Returns 181 on my laptop."
```

"Test of concatenation. Implementation using internal stream"

```
Time millisecondsToRun: [  
  | string stream |  
  string := 'abcdefg'.  
  stream := WriteStream on: (String new: 8000).  
  stream nextPutAll: string.  
  1000 timesRepeat: [stream nextPutAll: 'abcd']] "Returns 5."
```

The reason why concatenation is very inefficient is that it creates a new string containing a copy of the original and then adds the argument string to it. Don't use concatenation if you must repeat it more than a few times and if execution speed is important.

Main lessons learned:

- Internal streams are used mainly (but not exclusively) for operations on strings.
- Class `TextStream` adds emphasis handling to inherited character-oriented text operations.
- Appropriate use of internal streams makes programs simpler and often more efficient.

Exercises

1. Implement the problem in Example 2 with `Collection` methods and compare the two solutions.
2. Implement the problem in Example 2 with `String` methods and compare the two solutions.
3. Can you implement Examples 2 and 3 with class `StringParameterSubstitution`? Note that this implementation of string replacement is also based on streams.
4. Explain the definition of `printString` with your current background on streams.
5. What will happen if you open a `ReadStream` and a `WriteStream` over the same collection and use the two streams alternatively?
6. Browse uses of `ReadWriteStream`.
7. What happens when you execute `nextPut`: after reaching the last element and the underlying collection is not large enough?
8. Arrays cannot grow or shrink. What happens when you add an element to a stream whose underlying collection is an array?
9. Write method `skipSeparators: aCollection` to skip all elements included in `aCollection`.
10. The Transcript - an instance of `TextCollector` - is a major application of internal streams. In essence, a `TextCollector` is a value holder for the Transcript window and its contents are accessed via a write stream. This is why some parts of the Transcript protocol are identical to the protocol of internal streams. Write a short description of `TextCollector` focusing on its relation to internal streams.

10.4 Example: A Text filter

In Examples 2 and 3 in the previous section, we needed to replace strings, sometimes obtained by evaluating a block. In other words, we needed to filter input text and transform it into new text. This seems like a generally useful functionality and we will now implement it as a new class called `TextFilter`.

Specification: Class `TextFilter` takes an initial `String` object and replaces occurrences of any one of matching substrings with a corresponding `String` or `Text` object. Replacement objects are specified as `String` or `Text` objects or as blocks that calculate `String` or `Text` objects.

Examples of application:

- A form letter could contain 'formal parameters' (in the terminology of `StringParameterSubstitution`) such as '<name>' and '<date>', and the filtering process would replace the first parameter with a string provided by the user, and the second by an expression calculating today's day.
- A text editor could provide an extended string replacement facility allowing the user to replace not just one string but any number of strings simultaneously.

Scenario

Assume original string = 'abcdefg' and match/replacement pairs pair1 = 'bc'->'xxx', pair2 = 'bed'->'y'.

1. Set current position in string to 1. Compare \$a with the first character of pair1 key (no match) and first character of pair2 key (no match).
2. Increment position in string, compare with first character in both pairs, find match in both.
3. Increment position in string, compare with second character in both pairs, find match in both. pair1 match is complete, perform replacement, reset matching for both pairs.
4. Increment position in string, compare with first character in both pairs, and so on.

Preliminary Design: The specification can be implemented with a class-tool and the only questions are how to represent the necessary parameters and how to perform the replacement. We will implement the replacement by scanning the given string character by character and matching it against all match strings at each step. When a match is found, the corresponding replacement is made and the search continues from the next character of the original string. All partial matches are reset at this point.

Considering this principle, we immediately see that the state of processing and additional parameters require the following information:

- The original string and our current place in it.
- The new string as constructed up to this point and our current place in it.
- A collection of match strings and their replacements (strings, texts, or blocks)
- For each match/replacement pair, remember currently reached position in matching.

Design Refinement: We will now decide on the details of the components identified in Preliminary Design, and construct the replacement algorithm.

- The original string is accessed one-element-at-a-time and we will access it through a `ReadStream`. This also takes care of keeping track of the current location in the string.
- For the same reason, we will access the new string through a `WriteStream`.
- The obvious storage for strings and their translations is a dictionary with the match string as the key and the replacement string as the value. When we consider that we must also keep track of how much of the match string has been checked, we decide to hold this information also as a part of the value. Altogether the dictionary elements will be match string -> Array (replacement value, position).

The replacement algorithm will be as follows:

1. Create a `ReadStream` over the original string and a `WriteStream` over the string being constructed. Initialize the second element of the value array of each element to 0.

2. Repeat for each position of the input stream beginning from the start:
 - a. For each element of the dictionary do:
 - i. Increment current position holder for match string.
 - ii. Compare input string and match character.
 1. If no match, reset current position holder for match string to 0.
 2. If match, check if this is the last character to match.

If this is the last character (match succeeded), make replacement in output stream, reset current position holder for match string in all dictionary entries to 0, and repeat Step a.

If this is not the last character (match incomplete), increment current position holder.

The intent is to perform filtering is a one-step operation – by submitting a string with all filter parameters, executing the message without interruption, and receiving the result. We will thus never need more than one instance of the filter at a time and we will implement the method as a class method, somewhat like sort: in `SequenceableCollectionSorter`³.

We now have all necessary information except for the placement of `TextFilter` in the class hierarchy. Since there are no related classes, we will make `TextFilter` a subclass of `Object`.

Implementation:

The comment of `TextFilter` is as follows:

I implement general filtering of text. To create an instance, I need the original string and two arrays consisting of strings to be matched, and replacements. Replacement values may be string or text objects or blocks. My filtering method returns the result without affection the original.

Class Variables:

<code>InputStream</code>	<code><readStream></code>	streams over input string
<code>OutputStream</code>	<code><writeStream></code>	used to build filtered string
<code>MatchDictionary</code>	<code><String, Text, Block></code>	used to do replacements

`TextFilter` will implement all its functionality via class method `filter: aString match: matchArray replace: replacementArray`. The definition strictly follows the algorithm outlined above but we will restrict our implementation to string replacements and leave extension to `Text` and `BlockClosure` arguments as an exercise. The definition is as follows:

filter: aString match: matchArray replace: replacementArray

```
"I filter aString using matchArray and replacementArray, and return the resulting String ."
"Initialization."
MatchDictionary := Dictionary new.
matchArray with: replacementArray do:
    [:match :replace | MatchDictionary at: match put: (Array with: replace with: 0)].
InputStream := ReadStream on: aString.
OutputStream := WriteStream on: (String new: aString size).
"Filtering."
[InputStream atEnd] whileFalse: [self matchAndReplace].
^OutputStream contents
```

Most of the work is done by class method `match` which takes a single character from the input stream and tries to match it. Its definition is

matchAndReplace

```
"Get next character, match it against all dictionary entries, and do replacement if necessary."
| ch |
```

³ Defining behavior via class methods is generally frowned upon by Smalltalk experts because it may complicate specialization via subclassing. In our example, we are following the philosophy of the sorting mechanism in class `SequenceableCollectionSorter` which serves a similar purpose.

```
ch := InputStream next.  
"Copy the input character into the output stream for now."  
OutputStream nextPut: ch.  
"Now try to match against successive entries in the dictionary."  
MatchDictionary  
  keysAndValuesDo:  
    [:key :value |  
    | index |  
    "Get index of next character in this dictionary entry."  
    index := (value at: 2) + 1.  
    "Check if it equals the input character."  
    ch == (key at: index)  
      ifTrue: [index = key size "We have a match. Did we match the whole  
                                replacement value?"  
              ifTrue: "We matched the whole value."  
                    "Go back in output stream for replacement."  
                    [OutputStream skip: key size negated.  
                    "Put replacement into output stream."  
                    OutputStream nextPutAll: (value at: 1).  
                    "Reset match positions in all entries."  
                    MatchDictionary do: [:valueArray |  
                                          valueArray at: 2 put: 0].  
                                          "Done with this character."  
                                          ^self]  
              ifFalse: "Not end of matching yet - update index."  
                      [value at: 2 put: index]]  
      ifFalse: "No match, reset index in this entry to 0."  
              [value at: 2 put: 0]]
```

This seems a bit long but that's mainly because of our copious comments. To test the method, I executed the following test code

```
|matchArray replacementArray|  
matchArray := #('ab' 'eab').  
replacementArray := #('xx' 'yy').  
TextFilter filter: 'abcdeab' match: matchArray replace: replacementArray .
```

with *inspect* and got 'xxcdexx' which is not quite what I expected - I hoped for the 'better' match 'xxcdyy'. (Essentially, by 'better' I mean 'more compressed'.) What is the problem?

In fact, the problem is with our specification. What is happening is that in our example that the 'ab' -> 'xx' replacement is made before the method can make the nicer 'eab' -> 'yy' replacement. We should have said that if several replacements are possible in a given pass, one of those that give the longest replacement will be made. Implementing this specification would have produced the 'expected' result. We will formulate a better specification and develop a solution in the next chapter.

Could we have avoided our mistake? If we executed a scenario corresponding to our example in its entirety, we would have noticed the problem. The conclusion is that not only the implementation but also the design and even the specification must be tested.

Main lessons learned:

- When we know that we will never need several instances of a class, we can implement its functionality as a class protocol.
- Before you conclude that your design is incorrect, make sure that your specification is correct and complete. Better still, make sure that your specification is correct before you start design. A good way to obtain this assurance is to completely execute a set of exhaustive scenarios.

Exercises

1. Extend `TextFilter` to accept blocks as replacement arguments as stated in the specification.

10.5 Example: Circular Buffer

In computing terminology, a *buffer* is a memory area that accepts data from one process and emits it to another process; the two processes work at their own speeds. An example of the use of a buffer is reading a block of data from a file into memory where it is processed one byte at a time by a program. Another example is a computing node on a network that accepts parcels of data arriving in unpredictable amounts and at unpredictable times, processes them one byte at-a-time, and possibly sends the data on to another network node.

The hardware implementation of buffers often has the form of a special memory chip with a fixed number of memory locations with pointers to the first byte to be retrieved, and to the location where the next byte is to be stored as in Figure 10.8. When a new byte arrives, it is stored at the next available location and the pointer is incremented, and when a byte is required from storage, it is removed from the location pointed to and the pointer incremented.

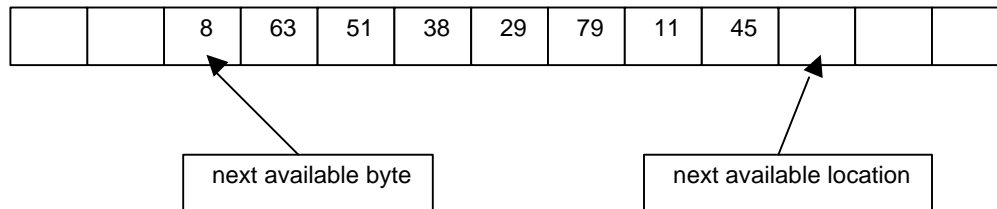


Figure 10.8. Buffer as a fixed size array with pointers to the next available byte and the next available location.

In reality, of course, a byte read from the buffer is not ‘removed’ and only the changed value of the pointer indicates that the byte has been used. Similarly, an ‘empty’ position is not really empty but the new value simply overrides the old value. Finally, when a pointer reaches the end of the array, the buffer is treated as if it were circular, as if its beginning were glued to its end, and when the pointer reaches the end, it ‘increments’ by being repositioned to the start. In mathematical terms, incrementing is performed in modular arithmetic as the remainder of division of the position by the size of the buffer.

The buffer does not, of course, have to be a special hardware chip and, in fact, it usually is not. Instead, it can be just a memory region that emulates the circular buffer area. Implementing this pretend circular buffer structure is the purpose of this section.

Problem. Implement a circular buffer based on a fixed-size array. Instances of the `CircularBuffer` class implementing this structure must be able to return the next available byte as a result of executing the `next` message which also updates the internal pointer, and to store a byte in response to `nextPut:`, again automatically updating the internal pointer. The buffer can also be tested with messages `isEmpty` and `isFull`.

Solution. If it wasn’t for the very suggestive choice of message names, our first impulse would probably be to implement `CircularBuffer` as some kind of collection. On second thought, it becomes clear that `CircularBuffer` is not a collection but rather a mechanism for accessing the collection hidden inside it. Since the access is ‘linear’, this immediately suggests that `CircularBuffer` is a stream. We will thus implement it as a part of the `Stream` hierarchy.

The next question, of course, is where to put it in the `Stream` tree. To answer this question, let’s start from the top and go down only as far as necessary to inherit useful behaviors. Class `PeekableStream` adds the ability to peak ahead but if we emulate the model of a hardware chip, such functionality should not be present and we conclude that we should subclass `CircularBuffer` directly to `Stream`.

The last question before we start implementing the class is what functionality it should implement. According to the specification, we need an accessing protocol (`next`, `nextPut:`), a testing protocol (`isEmpty`,

isFull), and it will be useful to implement enumeration (message do:) for consistency with other streams and for printing. A printing protocol is necessary for the inspector and for testing, and initialization is required to back up the creation protocol. Printing obviously enumerates over all elements in the buffer and we will thus need an enumeration protocol. With this, we can now start implementing the class.

The class will need an instance variable for the array that holds the data (array), pointers to the first available location and the first available element (firstLocation and firstElement), and it will be useful to have a variable to hold the state (isEmpty). The modular arithmetic that we will need for updating indices will require the size of the underlying array. We will keep it in an instance variable so that we don't have to retrieve it every time and since the buffer may not be full at all times, we will refer to it as capacity.

The creation message will create an instance with an array of the specified size and initialize the remaining instance variable

new: anInteger

```
^self basicNew initialize: anInteger
```

where

initialize: anInteger

```
array := Array new: anInteger.  
capacity := array size.  
firstIndex := 1.  
lastIndex := 1.  
isEmpty := true
```

initializes the instance variables in an obvious way. A simple test such as

```
CircularArray new: 10
```

executed with *inspect* confirms that everything is OK so far.

What should we implement next? We cannot do anything without nextPut: and next, and these require testing for empty and full so we will first implement the testing methods. Method isEmpty simply returns the value of isEmpty but isFull requires calculation. The buffer is full if the firstLocation has been pushed far enough to coincide with firstElement and so

isFull

```
"Are all slots occupied?"
```

```
^(firstElement = firstLocation) and: [isEmpty not]
```

because the two pointers will coincide not only when the buffer is full but also when it is empty.

With these two methods, we can now implement next and nextPut:. Method nextPut: adds a new element if the buffer is not yet full. After the test, it then puts the new element into the first available location and updates the pointer:

nextPut: anObject

```
"Add new element if there is room, otherwise execute exception block."
```

```
self isFull ifTrue: [^self error: 'Buffer is full'].  
array at: firstLocation put: anObject.  
self moveFirstLocationIndex.  
^anObject
```

Moving of the first location pointer is left to another method which increments the pointer using modular arithmetic and adds 1 because modulo n arithmetic counts from 0 to n-1 whereas arrays are numbered from 1 to n-1:

moveFirstLocationIndex

```
"An element has been added, 'increment' firstIndex."
```

```
firstLocation := (firstLocation rem: capacity) + 1.
```

```
isEmpty := false
```

Method next first checks whether the buffer is empty and if it is not, it returns the element at the pointer location and updates the pointer:

```
next
"Return next element and move pointer, return nil if empty."
^isEmpty
  ifTrue: [nil]
  ifFalse:
    [| el |
     el := array at: firstElement.
     self moveFirstElementIndex.
     el]
```

Here incrementing is done with modular arithmetic as follows:

```
moveFirstElementIndex
"Element was removed, update firstElement."
firstElement := (firstElement rem: capacity) + 1.
isEmpty := firstElement = firstLocation
```

Finally, we can now implement printing, in other words method printOn: aStream. The desired format is

```
CircularBuffer (13 25 11)
```

which hides how the data is arranged internally and shows the first element to be retrieved next as the first element inside the brackets, in this case 13. The definition is simple

```
printOn: aStream
"Append to the argument aStream a sequence of characters that identifies the collection."
| first |
aStream print: self class; nextPutAll: ' ('.
first := true.
self do: [:element | first ifTrue: [first := false]
                           ifFalse: [aStream space].
                           element printOn: aStream].
aStream nextPut: $)
```

if we have a do: message that processes the elements starting with the first available element and ending with the last one. This operation is implemented as follows:

```
do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
self isEmpty ifTrue: [^self].
firstElement >= firstLocation
  ifTrue:
    [firstElement to: capacity do: [:index | aBlock value: (array at: index)].
     1 to: firstLocation - 1 do: [:index | aBlock value: (array at: index)]]
  ifFalse: [firstElement to: firstLocation - 1 do: [:index | aBlock value: (array at: index)]]
```

The principle of this method is that if the buffer is not empty, the index of the first available element is either less then the index of the last available element or the opposite is true (Figure 10.9). The handling of these two cases can be deduced from the diagram.





Figure 10.9. The two possible relative positions of firstElement and firstLocation. Arrows point from first available element upward. Filled circle denotes the first element, filled square is the last element.

Exercises

1. We have cached the value of size and isEmpty in instance variables to avoid the need to recalculate them. Is there any advantage in caching isEmpty? Implement this modification and note that this internal change has no effect on the behavior of CircularBuffer or any other classes the use it.

10.6 Introduction to files and external streams

External streams are the basis of operations on files and all other data transmission that occurs as a stream of bytes such as network data transmission. We will focus on the use of external streams with files which allows operation on textual data, graphics, sound, and other digital information. In this section, we introduce the basics of file and external streams, and several examples of their use are presented in the following sections.

Smalltalk operations on files and directories are implemented by combining external streams and class Filename (Figure 10.10). The main purpose of *external streams* is to provide byte-oriented streaming access to data, the role of *Filename* is to construct filenames, allow checking whether a file exists, whether a filename name has the proper structure, creating a new directory or file, and perform other file-system related operations.

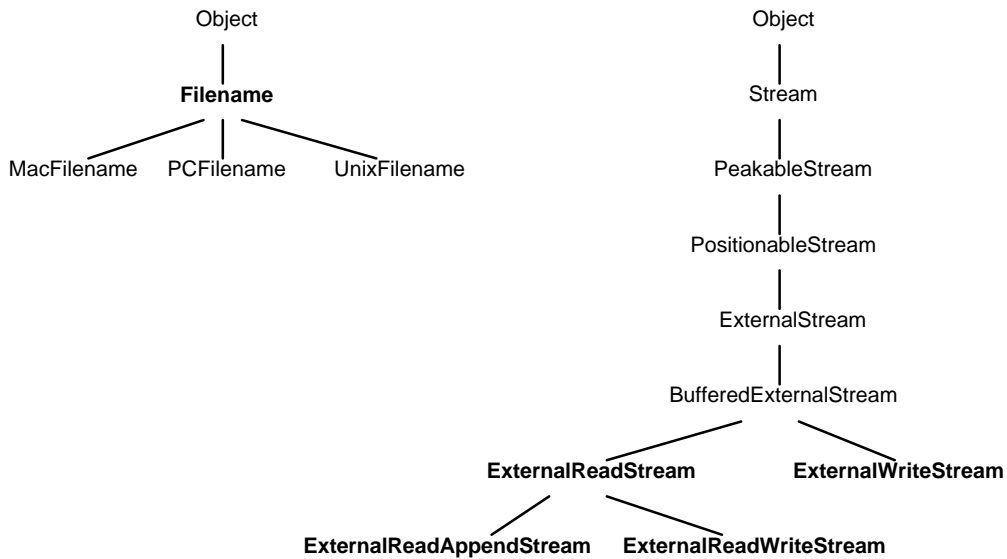


Figure 10.10. Main classes used in file processing.

The cookbook procedure for processing data stored in a file is as follows:

1. Create a **Filename** object with a filename string. The string is the name of the file that may include its drive/directory path.
2. Create the appropriate kind of external stream and associate the **Filename** object with it.
3. Perform byte operations on to the stream.
4. Close the stream object; this will close the file too.

Closing a file is very important for two reasons. One is that if a file is not explicitly closed, the data 'written to it' may not be stored on the disk. The second reason is that the operating system assigns to each file that it opens one of a limited number of 'handles'. Failure to close a file means that the handle is not released and if too many handles are in use, new files cannot be open. It may then be impossible even to save your work when leaving Smalltalk!

As a preliminary example demonstrating the above procedure and the role of external streams and **Filename** objects, the following code fragment opens a file for writing, stores some information in it, and closes the stream and its associated file.

```
[file fileStream]
file := Filename named: 'c:\testfile'. "Open a file called 'testfile' in the root directory on drive C."
fileStream := file writeStream. "Attach the file to a write stream (write only access)."
fileStream nextPutAll: 'abc'. "Store the ASCII codes of 'abc' in the file buffer."
fileStream close "Flush buffer to disk and release OS handle."
```

Execute the program and open the file with the file editor to see that the file has indeed been created and contains the string 'abc'.

Although almost all Smalltalk applications use files, direct byte-oriented operations on files via streams as shown above are rare (except when reading data, possibly coming from a network) because **Filename** and external stream operations cannot directly store *objects*. Smalltalk programmers thus use files and external streams mainly as a vehicle for more powerful object-oriented tools such as BOSS (Section 10.9) and for operations on directories or files as a whole.

Class **Filename** and various external stream classes contain a large number of methods and we will present only the most important ones. Before we do, however, a few comments about the classes themselves.

Class *Filename* is an abstract class and its concrete subclasses (*MaxFilename*, *PCFilename* and its subclasses, and *UnixFilename*) implement the platform-specific behavior needed on your machine, such as parsing platform-specific syntax of file names. However, you never need to deal with these concrete subclasses because *Filename* automatically sends all platform-dependent messages to the subclass representing your platform. This is done via *Filename*'s class variable *DefaultClass* which holds the name of the appropriate *Filename* class. Removing explicit dependence on one platform makes it possible to write programs that will run on different platforms. This arrangement is similar to the implementation of strings.

External streams perform data transfer operations. Instances of external streams are never created by class messages to external stream classes but by messages to the *Filename* object as in our example above. The *Filename* object, in turn, asks class *ExternalStream* to create and return the appropriate kind of stream; this procedure also opens the file.

After this brief introduction, we will now introduce class *Filename* and its essential protocols. We will then present external streams.

Main lessons learned:

- Byte-oriented file operations require the combination of *Filename* and an external stream.
- *Filename* objects provide interaction with the file system, external streams provide byte-by-byte access to file elements.
- *Filename* is an abstract class which transparently communicates with appropriate concrete subclass.
- Smalltalk programs rarely perform byte-oriented file access explicitly. To store and retrieve objects in files, use tools such as BOSS or a data base program.

Exercises

1. Examine and describe how *Filename* achieves passing of messages to its concrete subclass. Compare this with the similar behavior of *Character*.
2. We created a write stream by sending *writeStream* to *Filename*. Examine its definition.

10.7 Class *Filename*

Class *Filename* is an interface to the file system and provides access to files and directories. The essence of its comment is as follows:

Class *Filename* is an abstract class. Instances of its subclasses encapsulate the platform-specific syntax of OS file path names. This class can almost be used as a concrete class, except name syntax is not interpreted. There is standard protocol provided to do most of the things that OS's can do with references to files -- deleting, renaming, etc.

The best way to understand the role of *Filename* is to examine its protocols and file-related protocols in other classes.

Creating *Filename* objects

Filename objects can be created in two ways:

- By sending *named: aString* to *Filename* as in *Filename named: 'prog'*.
- By sending *asFilename* to a string as in *'c:\st\prog' asFilename*.

In both cases, the string may be either a 'relative' specification (the first example) or an 'absolute' specification (the second example). In other words, the filename string may refer either to a file in the currently active directory or specify the complete path. As another example of relative specification, *'file.st'* refers to the file called *'file.st'* in the current directory, whereas *'c:\smalltalk\examples\example.1'* specifies

the path including the disk drive. Certain messages (but not creation messages) allow wildcard characters # (any single character) and * (any group of characters) inside a filename. As an example, in some contexts 'story.1#' refers to any string consisting of 'story.1' followed by a single character (such as 'story.12'), whereas 'story.1*' refers to any string starting with 'story.1' followed by zero or more characters (as in 'story.1', 'story.12' or 'story.123').

If you are developing an application that should run on several different platforms, you must consider that different operating systems use different separators between directories and file names in the filename path (in our example, we assumed the PC platform which uses \). To get the appropriate separator for an arbitrary platform, use message `Filename separator`; this way, the program can construct path name at run time in the appropriate way. Remember, however, that different platforms also have different rules for the maximum filename length. To get the maximum filename length for a platform, execute `Filename maxLength`. You can ignore these details if your application is designed to run on one platform only.

Filename prompts in class Dialog

Class `Dialog` provides several powerful `requestFileName:` messages in the *file name dialogs* protocol. All these methods prompt the user for a file name and return a *string* which can then be used to construct the `Filename` object as explained above. These messages also allow you to specify, for example, whether the file should be new (succeeds only if the file does not yet exist) or old (succeeds only if the file already exists). Some of these messages repeat prompting until the desired condition is satisfied, and some allow you to specify a block to be executed when the message fails. The simplest of these messages is `requestFileName:` which displays a prompt . It can be used as in

```
| file |  
file := (Dialog requestFileName: 'Enter file name') asFilename
```

This message behaves just like the familiar `Dialog request:` but allows wildcard characters * and # in the answer. If the user enters a string with wildcard characters, the method displays a pop up menu containing the names of all matching filenames and allows the user to make a selection, try again, or abort by clicking *Cancel* (Figure 10.11). In the last case, the message returns an empty string and this must be kept in mind to prevent `asFilename` from crashing.

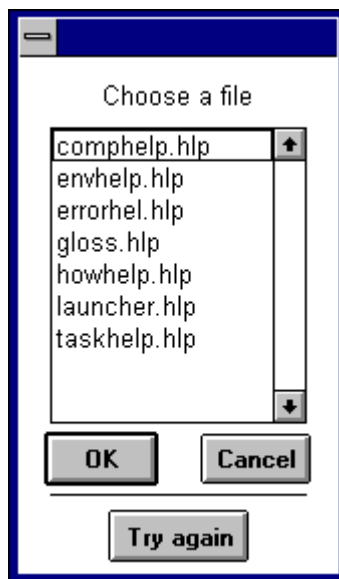


Figure 10.11. Possible result of typing '*.hlp' in response to `Dialog requestFileName:`.

The proper use of the combination of `requestFileName:` and `asFilename` should thus be something like

```
| file name |  
name := Dialog requestFileName: 'Enter file name'.  
name isEmpty ifTrue: [^self].  
file := name asFilename.  
etc.
```

A more powerful filename dialog message has the form `requestFilename:default:`. This message allows the specification of an initial filename as in

```
Dialog requestFileName: 'Select a file' default: '*.st'
```

An even more powerful version is `requestFileName:default:version:` which lets you specify not only the default filename but also its type. The `version:` argument may be `#mustBeNew` (user is asked how to proceed if the filename already exists on the specified path), `#mustBeOld` (user is asked what to do if the name is not found on the specified path), `#new` (user is warned if the file exists), `#old` (user is warned if the file does not exist), or `#any`. Yet another version of file prompt messages is the message `requestFileName:default:version:ifFail:` which includes an exception block to be executed when the 'version' condition fails in the case of `#mustBeNew` or `#mustBeOld`.

Accessing operations are scattered across several protocols and include the following methods:

`contentsOfEntireFile` - opens an external read stream on the file, gets its contents, returns it as a `String`, and closes the stream and the file. The user is not aware of the read stream created and closed during the operation. Note that we can also access the contents of a file by attaching it to an external stream and sending contents to the stream. However, message `contentsOfEntireFile` saves you from creating an external stream and closing it explicitly. The following example creates a new file, stores some data in it, closes the file, and gets and displays its contents.

```
|file fileStream|  
"Create a file, put some text in it, and close it."  
file := Filename named: 'c:\testfile'.  
fileStream := file writeStream.  
fileStream nextPutAll: 'abc'.  
fileStream close.  
"Display file contents in the Transcript"  
Transcript cr; show: (Filename named: 'c:\testfile') contentsOfEntireFile "Displays the string 'abc'."
```

`directory` - returns the directory containing the file corresponding to the `Filename` receiver. As an example,

```
| file |  
file := 'c:\abc\xyz' asFilename.  
file directory
```

returns an object such as a `FATFilename` (a concrete subclass of abstract class `PCFilename` for the MS-DOS operating system). Note that both files and directories are instances of `Filename`.

Class message `defaultDirectoryName` returns the *String* describing the full path of the current directory as in

```
Filename defaultDirectoryName " Returns, for example, 'c:\visual\image'"
```

The related class message `currentDirectory` returns the corresponding *Filename* directory object.

Deleting, copying, moving, renaming, and printing files

delete - as in fileName delete - deletes the Filename object (a file or a directory). As an example of its use, the following fragment creates and opens a file called 'test' in the root directory of drive C, stores data in it, closes it, displays the file's contents, and deletes the file:

```
|file fileStream|
"Create, initialize, and close a file."
file := Filename named: 'c:\testfile'.
fileStream := file writeStream.      "Create write stream on the file."
fileStream nextPutAll: 'abc'.
fileStream close.
"Display file contents in the Transcript."
Transcript cr; show: (Filename named: 'c:\testfile') contentsOfEntireFile.
"Delete the file."
(Filename named: 'c:\testfile') delete
```

Note that delete must be sent to the Filename object - the stream does not understand delete.

renameTo: pathName renames the receiver Filename object, *and moves it* to a new directory if the new path is different from the old one; the original name is deleted. As an example,

```
| filename |
filename := Filename named: 'test'.
filename renameTo: 'c:\smalltalk\examples\example.1'.      "Renames and moves the file."
```

copyTo: pathNameString creates a copy of the receiver under a new name, possibly in a new location. The original file and its name remain unchanged.

To *print* a text file, print its String contents. To print a PostScript file, use class Document.

Testing

exists - checks whether the receiver Filename exists and returns true or false. Note again that the receiver may be a file or a directory. As an example,

Filename defaultDirectoryName asFilename exists

returns true.

isDirectory - tests whether the Filename receiver is a directory or a file. Returns true for a directory, false for a file. As an example,

Filename currentDirectory isDirectory

returns true.

Directory operations

makeDirectory - creates a directory according to the specification in the Filename receiver as in

(Filename named: 'new') makeDirectory	"Creates subdirectory 'new' of the current directory."
(Filename named: 'c:\dos\new') makeDirectory	"Creates directory 'new' with the specified path."

dates returns an IdentityDictionary containing the dates of creation, last modification, and last access of the receiver - if these parameters are supported by the operating system. As an example,

| file |

file := (Dialog requestFileName: 'Enter file name' default: '*.st') asFilename.

file dates

returns an instance of IdentityDictionary with date information on a file selected by the user. On PC platforms, for example, this fragment will return something like

IdentityDictionary (#statusChanged->nil #modified->#(6 April 1993 1:59:50 pm) #accessed->nil)

where nil values indicate that the corresponding parameter is not supported on the current platform.

directoryContents returns an array of strings, the names of files and subdirectories in the current directory. As an example,

Filename currentDirectory directoryContents

could return something like #('VISUAL.IM' 'VISUAL.SOU' 'VISUAL.CHA' 'WORKSP.2')

Main lessons learned:

- Class Filename supports operations such as deletion, renaming, copying, and closing of files and directories. It also provides tests and access to internal parameters such as the length of a file and the contents of a directory.
- Some file operations require only Filename, others also require an external stream.
- Filename can be used to create directories but creation of files requires an external stream.
- Class Dialog provides several file-related dialogs that search the directory for the specified filename, allowing wildcard characters and specification of the type of the desired file.

Exercises

1. Try requestFileName:default:version: with various values of version: and names of files that already exist/don't yet exist.
2. What happens when you send message directory to a Filename object and the directory with the specified name does not exist?
3. Create a table of all essential messages introduced in this section. For each method, specify whether it is a class or an instance method, what are its arguments, what object it returns, what are its preconditions, and what is its effect.
4. Why are defaultDirectoryName and currentDirectory class methods?
5. Define method deleteDirectory: aString ifFail: aBlock which checks whether aString is a directory name, deletes the directory if appropriate, and executes aBlock otherwise.

10.8 Examples of the use of file operations that don't require external streams

External streams are needed only for byte-by-byte access to files. Operations on directories, and operations on the contents of a file treated as a string do not require explicit use of external streams. This section gives several examples of such operations.

Example : List alphabetically all files in the current directory and their sizes

Solution: As we know, there are two messages to access to current directory. Message currentDirectory returns the current directory as a Filename object, and message currentDirectoryString returns a String containing the filename path of the current directory. Since we need the contents of the file, we need the Filename object. We will thus use the currentDirectory message.

If you examine the Filename protocols, you will find that you can get the contents of a Filename directory object by sending it the directoryContents message. This message returns an Array of strings - names of the files and subdirectories in the receiver - and to sort it alphabetically, we will convert it to a SortedCollection. To obtain information on the corresponding files, we must create Filename objects over the individual string elements and ask them about their size using the instance message fileSize (returns the size of the file in bytes). The whole program is as follows:

```
| names |  
"Extract names and convert to sorted collection."  
names := Filename currentDirectory directoryContents asSortedCollection.  
Transcript cr.  
"Convert names individually to filenames and extract and print the desired information."  
names do: [:name | Transcript show: name; tab; show: name asFilename fileSize printString; cr]
```

Note that the program does not check whether the extracted names are names of files or directories and lists them all. We leave it to you to correct this imperfection.

Example 2: Test whether two files (two directories) entered by the user have the same contents

Solution: To check whether two files contain the same data, we don't need an external stream because we can compare the contentsOfEntireFile of both files:

```
| file1 file2 text1 text2 |  
"Let the user select two files from the current directory."  
file1 := (Dialog requestFileName: 'Select the first file.' default: '*.*)') asFilename.  
file2 := (Dialog requestFileName: 'Select the second file.' default: '*.*)') asFilename.  
text1 := file1 contentsOfEntireFile.  
text2 := file2 contentsOfEntireFile.  
text1 = text2
```

If the files are large, this program will work with two large objects and take a long time to execute. Using streams explicitly may then be preferable.

Example 3: Let user delete a file from a list

Problem: Implement a method to display the file names in the current directory in a multiple choice dialog, and allow the user to delete a file

Solution: This problem does not require a specific Filename and we will implement it as a class method in Filename, following the example of several existing *fromUser methods. The method will obtain the current directory, display its contents in a multiple choice dialog asking the user which file to delete, and delete the file if the user makes a selection. The implementation is as follows:

deleteFromUser

“Display dialog with names in current directory and allow user to delete one.”

| choice fileNames |

“Display dialog with names of all files in the current directory.”

fileNames := Filename currentDirectory directoryContents asSortedCollection.

choice := Dialog

choose: 'Which file do you want to delete?'

fromList: fileNames

values: fileNames

lines: 20

cancel: ["].

“If the user selected a file, delete it.”

choice isEmpty ifFalse: [choice asFilename delete]

Main lessons learned:

- File and directory operations that don't require explicit byte-oriented access can be performed without external streams. These operations include operations on entire contents of a file, deleting, renaming, accessing contents, and similar operations.

Exercises

1. Refine Example 1 to distinguish between files and subdirectories. Your version of the program should print 'directory' instead of the size for those filenames that are directories.
2. Define a new method called = to test whether two files or directories have the same contents.
3. Why does Example 2 take so long to execute for larger files?
4. The method in Example 3 is not a safe way to delete files and it does not distinguish between files and directories. Write a new version that will request a confirmation and ask the user whether to delete a subdirectory if it is not empty.
5. When you use named: to create a new Filename object on a PC platform, the name is reduced to at most 8 characters. How does this happen? Since this can be a problem with names of drives on networks, can it be avoided? (Hint: Try another creation method, possibly inherited.)

10.9 External streams

We have seen that Filename and external stream functionalities somewhat overlap. If you find it confusing, the rule of thumb is that creating external streams and attaching them to Filename objects is necessary only to access the contents of the file in a streaming fashion or to store or access objects created by BOSS. External streams are not necessary for operations on whole files and directories.

We have already seen that the attachment of a stream to a file is performed by asking a Filename to create a stream of the desired kind. The following messages are available for this purpose: appendStream, newReadAppendStream, newReadWriteStream, readStream, readAppendStream, readWriteStream, and writeStream. Each of them creates a different kind of stream over the same file and will now explain them briefly. A summary table is provided below.

Creating external streams

- **appendStream** opens an 'append stream', a file that allows only sequential writing at the end. As an example, create file 'test' containing the string 'abc' using the file editor. The following program

```
|file fileStream|
file := 'test' asFilename.
fileStream := file appendStream.      "Attach file to an append stream."
fileStream nextPutAll: 'xyz'.          "Store the ASCII codes of 'xyz' in the file buffer."
fileStream close                       "Close file via its associated stream."
```

opens the file for appending, writes the three characters 'xyz' at the end, and closes the file. The file now contains 'abcxyz'. Check this by opening an editor on the file.

- **newReadAppendStream** opens an ExternalReadWrite stream at the beginning of the file for unrestricted reading, but writing is restricted to appending at the end. For reading, the file can be positioned with position: but this has no effect on writing. The message clears (erases) all original contents if the file already exists; this is suggested by the word new in the name of the method.
- **newReadWriteStream** opens a new read/write stream that can be randomly positioned for both reading and writing using position:. The word new in the name of the method indicates that if the file existed before the message was sent, the original contents are deleted. Writing to a position within the file replaces the old byte with the new value.
- **readAppendStream** has the same properties as newReadAppendStream but does not reset and clear the receiver file.
- **readStream** can only read an existing file and is fully positionable.
- **readWriteStream** opens a read/write stream on a new or existing file without deleting the old contents. This is indicated by the absence of new in the name of the message. The message opens the file at the beginning and allows arbitrary positioning. It behaves as newReadWriteStream in all other respects.
- **writeStream** opens a purely sequential write stream at the beginning of an existing or new file. If the file existed, all data is lost. The stream does not understand any positioning messages and cannot be read.

	readable	writable	positionable	append only	clears original
appendStream	no	yes	no	yes	no
newReadAppendStream	yes	yes	reading	yes	yes
newReadWriteStream	yes	yes	yes	no	yes
readAppendStream	yes	yes	reading	yes	no
readStream	yes	no	yes	n/a	no
readWriteStream	yes	yes	yes	no	no
writeStream	no	yes	yes	no	no

Table 10.1. Properties of external streams.

To understand how stream messages work, it is useful to examine the definition of newReadAppendStream which is as follows:

newReadWriteStream

"Answer a new readWrite stream connected to the file represented by the receiver."

```
^ExternalReadWriteStream on:
    (FileConnection
        openFileNamed: self
        mode: #readWrite
        creationRule: #truncateOrCreate)
```

This explains how the limited number of external stream classes (Figure 10.8) can provide such a variety of accessing modes - the type of access is controlled by an instance of `FileConnection`. The other stream creation messages are similar.

Since a file and its mode of access are two separate things, a file initially accessed via one kind of stream may be closed and accessed again via another type of stream. As an example, we have already seen that you may open a file for writing, store some data in it, close it, and open it for reading later.

Operations on external streams

The following are the main operations on external streams:

Accessing

Includes reading or writing of individual bytes to or from the buffer and control of the buffer itself. The operation of accessing messages depends on the principle of the interface between Smalltalk's external streams and the operating system, and between the operating system and disk storage. This will be explained next.

As we already mentioned, external streams are 'buffered' which means that the stream object holds on to the part of the file which it is currently accessing via its instance variable `ioBuffer`. When accessing operations fill the stream buffer, its contents are automatically sent to the operating system and the buffer is reset. You can also perform this operation explicitly by sending `flush`, `commit`, or `close` to the stream (see below). For read-only streams, the buffer is just a multi-byte window into a file stored on the disk.

In addition to the buffer kept by Smalltalk, the operating system maintains its own buffer which operates in a similar way but is under the control of the operating system. Sending the contents of the stream buffer to the operating system thus writes to the *operating system's buffer* but it does not guarantee that the contents of the buffer is written to the disk ('committed'). Messages `close` and `commit` perform even this task.

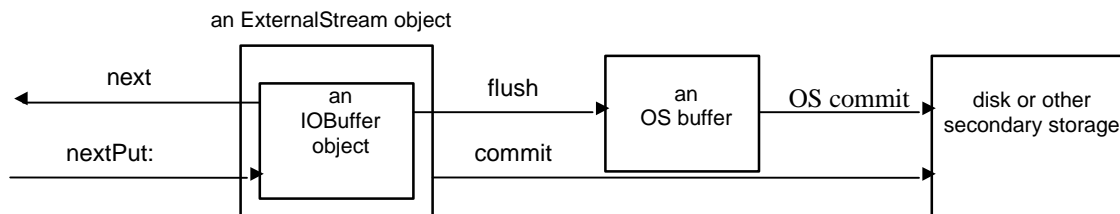


Figure 10.12. Data transfers resulting from various stream messages.

The buffer flushing, committing, and accessing messages are defined on writeable streams (Figure 10.12). Note again that these operations work on external streams, not on `FileName` objects! Note also that `flush` and `commit` are needed only for explicit buffer operations and that these operations happen automatically when the corresponding buffers become full.

- `flush` - sends the bytes accumulated in the stream's memory buffer to the *operating system*.
- `commit` - writes the contents of stream's buffer to the *disk*.
- `next`, `nextPut:` - streaming access in the style of `next` and `nextPut:` messages of internal streams. Operate on the contents of the stream buffer and flush or refill it when necessary.

We have already mentioned that external streams are normally read one character at a time; in other words, they are byte-oriented. They can, however, also be accessed in a bit-wise fashion. To access external streams one *bit* at a time, send message `binary` to the stream. To change bit access back to byte access, send `text` to the stream. Note that although you can change stream access from byte-oriented to bit-oriented and vice versa while the file is open. On the other hand, you cannot change the type of stream (for example from write only to read only); to do this, you must close the file and create the desired new kind of stream.

Positioning

- position, position:, setToEnd – work the same way as for internal streams.

In closing, we will now illustrate the difference between the various writeable external streams on a short example.

Example: Behavior of writeable external streams

In this example, we assume that the underlying file called 'test' already exists and contains the string '123456789'. Each example is executed from this initial state.

- After executing the following code fragment that uses **appendStream**

```
| stream |
stream := (Filename named: 'test') appendStream.
stream nextPutAll: 'abc'.           "Store characters $a, $b, $c at the end of the file."
stream close
```

the contents of the file become '123456789abc'. The new data have been appended at the end, the old data have not changed. Neither positioning nor reading are possible.

- When you change the previous program to use **newReadAppendStream** as in

```
| stream |
stream := (Filename named: 'test') newReadAppendStream.
stream nextPutAll: 'abc'.           " Store characters $a, $b, $c in the file ."
stream close
```

the contents of the file become '123456789' to 'abc'. The old data is thus lost. The stream can be positioned with position: but positioning affects only reading. Writing always occurs at the end of the file.

- With **newReadWriteStream**, writing erases the original contents of the file. The pointer can be repositioned for reading and for writing within the limits of the new contents.

```
| stream |
stream := (Filename named: 'test') newReadWriteStream.
stream nextPutAll: 'abc'.           "The stream now contains three characters $a, $b, $c."
stream position: 1; nextPut: $X. "Replaces the second element."
stream close
```

changes the contents from '123456789' to 'aXc', erasing the original contents. Message nextPut: issued after setting position to 1 overwrites the element in position 2 because the value of position is incremented before writing takes place.

- With **readWriteStream**, we don't lose the original contents of the file.

```
|stream |
stream := (Filename named: 'test') readWriteStream.
stream position: stream size. "Position at end."
stream nextPutAll: 'abc'.
stream position: 1.
stream nextPut: $X.
stream close
```

changes the original contents '123456789' to '1X3456789abc'. The stream is fully positionable.

- Finally, a **writeStream** allows positioning and writing starts from the beginning of the file. The original contents of the file are lost.

```
| stream |  
stream := (Filename named: 'test') writeStream.  
stream nextPutAll: 'abc'.  
stream close
```

changes the contents of the file to 'abc'.

Finally, note that we have been careful to close the file stream when it was no longer needed.

Main lessons learned:

- Several kinds of external streams can be opened by sending the appropriate stream creation message to the Filename object.
- External streams differ in the kind of access (read-only, write-only, read-write) and the kind of positioning (random, sequential only, append only) that they provide.
- Stream messages operate on a part of the file stored in a buffer. The contents of the buffer is flushed to the operating system or committed to the disk only when the buffer fills, when a flush or commit message is sent, or when the file is closed.

Exercises

1. How is it that the reading position of a ReadAppendStream can be changed but writing always occurs at the end of the file?
2. We have seen that different kinds of file access are obtained by collaboration with class FileConnection. Write a short description of this class.
3. For each task listed below, write the message that will open the file for the specified purpose assuming that the file is named 'test' and is stored in directory c:\binary.
 - a. Append new data to the end of the file.
 - b. Empty the file and write new data into it in sequential order.
 - c. Open the file, add data at the end, and read data anywhere in the file.
 - d. Open the file without losing the existing data and write new data anywhere in the file. No reading is anticipated.
 - e. Same as the previous situation but you want to be able to read the data randomly too.
4. One of the numerous suggestions for extensions of VisualWorks tools is adding *save* and *load* commands to the <operate> menu in the Workspace. Implement these extensions as described below. (Hint: Use the Resource Finder tool to examine the menu bar of the Visual Launcher to find how it opens a Workspace.)
 - a. Command *save* opens a file dialog and when the user *accepts*, the contents of the whole Workspace window is saved in the specified file. The *load* command works similarly but adds the contents of the file to the current Workspace contents.
 - b. Add command *save it* to save only the currently selected part of the Workspace.
5. Define an internal read-append stream that stores its contents in an external stream and resets itself when it reaches a specified size.

10.10 Storing objects with BOSS

BOSS - Binary Object Streaming Service - is a very important tool for converting most types of Smalltalk objects into compact binary representation that requires relatively little memory space. Although BOSS is used mainly to store objects in a file and retrieve them, it can also be used for other purposes such as sending objects across a network. BOSS is the essence of all programs that store data in a file.

BOSS is implemented by a group of Smalltalk classes in category System-Binary Storage. It is a very powerful tool that can, for example, store both classes and their instances, help converting from one version of a class to another, and read objects in the sequence in which they were stored or in random order.

In this section, we will limit ourselves to the simplest but most important use of BOSS - storing class instances and accessing them sequentially. For more sophisticated use, refer to User's Guide. The typical BOSS usage pattern is as follows:

1. Create an instance of class `BinaryObjectStorage` and open it on an external stream associated with a file.
2. Read the stored objects from the stream using `next` or write them to the stream using `nextPut`.
3. Close the BOSS object; this closes the file too.

The main BOSS protocols are as follows:

Creating and closing BOSS

The following two class methods are used to create BOSS objects and tie them to streams:

- `onOld: aStream` creates a BOSS object for reading the stream argument associated with an existing file or for appending to it. The stream must, of course, be capable of the desired type of access.
- `onNew: aStream` creates a BOSS object for writing to `aStream` starting at the beginning of the file. The file does not have to be new but will be treated as if it were. The stream must be capable of desired type of access.
- `close` closes the stream and the file associated with the BOSS object.

Accessing - reading and storing objects, changing position

BOSS is based on streams and its accessing messages are a subset of Stream accessing messages. The most important ones are

<code>next</code>	reads and reconstructs the next object from the BOSS stream
<code>nextPut: anObject</code>	increments the position pointer and stores <code>anObject</code> in the stream
<code>nextPutAll: aCollection</code>	stores all elements of a collection of objects, one after another
<code>position</code>	returns the current position in the stream
<code>position:</code>	changes the current position
<code>reset</code>	resets position to start
<code>setToEnd</code>	resets position to end
<code>atEnd</code>	tests whether the stream is positioned at the end

The following example shows how to use BOSS to *store* an object in a *new file* called 'c:\boss.tst':

```
| boss |
"Create a BOSS object."
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst') writeStream.
"Store object in file using BOSS."
boss nextPut: #('string1' 'string2').
"Close BOSS object and the file."
boss close
```

The following complementary program *reads* the object back and recreates it. Note that it is not necessary to specify that the object is an array, this information is recovered by BOSS. Using BOSS is thus simple and the only thing you must watch is to assign the retrieved objects to the correct variables when you read the objects back.

```
| array boss |
"Create a BOSS object."
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst') readStream.
"Read the array previously stored in the file."
array := boss next.
```



```
"Close the file."  
boss close
```

In most situations, you will use BOSS to store complete objects by a single operation rather than storing each component separately. As an example, store a whole collection as one object rather than storing the elements as separate objects one after another - and read it back without reconstructing it laboriously by enumeration. If, however, BOSS is used to access and frequently modify a large collection of objects, and if this access is not always sequential, you may want to store the objects one after another.

Another point to realize is that when inter-related objects are 'bossed' to one file, no duplication occurs and object relationships are preserved. This does not happen if the objects are stored in separate files. The following example illustrates the difference.

Example: Store network of inter-related objects in one file

Consider the simplest group of inter-related objects – two arrays sharing one element (Figure 10.13). Clearly, this group consists of three objects – the two arrays, and the shared fraction.

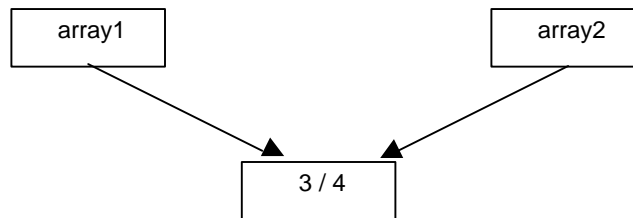


Figure 10.13. Two arrays sharing one element.

The following program creates the arrays, bosses them in two separate files, and reads them back. When you execute it with *inspect*, you will find that the test at the end returns false, indicating that the two reconstituted arrays *do not* share the fraction that the original arrays did (Figure 10.14). This is not surprising because we did not boss out any inverse references from the fraction to the other array.



Figure 10.13. The two arrays after bossing out into two separate files, and bossing in again.

```
| array1 array2 boss x |  
x := 3/4.  
"Create two arrays sharing one object and boss each to its own file."  
array1 := Array with: x.  
array2 := Array with: x.  
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst1') writeStream.  
boss nextPut: array1.  
boss close.  
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst2') writeStream.  
boss nextPut: array2.  
boss close.  
"Read the two objects back."  
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst1') readStream.  
array1 := boss next.
```

```
boss close.  
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst2') readStream.  
array2 := boss next.  
boss close.  
"Check whether the two arrays still share the element."  
(array1 at: 1) = (array2 at: 1) "Returns false."
```

If we now modify the program to write both arrays to the *same* file

```
| array1 array2 boss x |  
x := 3/4.  
"Create two arrays sharing one object and boss both to the same file."  
array1 := Array with: x.  
array2 := Array with: x.  
boss := BinaryObjectStorage onNew: (Filename named: 'c:\boss.tst1') writeStream.  
boss nextPut: array1.  
boss nextPut: array2.  
boss close.  
"Read the two objects back."  
boss := BinaryObjectStorage onOld: (Filename named: 'c:\boss.tst1') readStream.  
array1 := boss next.  
array2 := boss next.  
boss close.  
"Check whether the two arrays still share the element."  
(array1 at: 1) = (array2 at: 1) "Returns true."
```

we find that the two reconstituted arrays now *do* share the fraction, as they did before being bossed out. We conclude that storing multiple objects in one file preserves the original structure of their relationship. In fact, the second version also saves space because it stores the shared fraction object only ones.

In closing, we want to repeat that BOSS is one of the most important Smalltalk tools and if you don't have a data base program, you will probably store all your data using it. The small amount of space that we dedicated to BOSS is a tribute to the simplicity of its basic use and does not reflect its importance. We will use BOSS in all our applications to store persistent data.

Main lessons learned:

- BOSS - a collection of built in Smalltalk classes - is the standard tool for storing objects in files.
- BOSS is one of the most important VisualWorks tools.
- To use BOSS, create an instance of `BinaryObjectStorage` on a suitable external stream, perform the storage or retrieval operation, and close the BOSS object.
- Upon reading an object, BOSS recognizes its type automatically.
- Store compound objects as single entities rather than component by component.
- Store interrelated objects in the same file.

Exercises

1. What happens to the value of a variable associated with a `BinaryObjectStorage` when you close the BOSS object?
2. Must the stream used by BOSS be an external stream?
3. Write a program to use BOSS to store an array containing the factorials of all integers from 1 to 20 in file 'test' in directory c:\. Write another program to read the object back and print it in the Transcript.
4. Open a file editor on the BOSS file created in the previous exercise.
5. BOSS can be used as a simple database system by storing elements of a collection in consecutive locations and accessing them by position, for example through some translation table (a dictionary). Explain how this would be done on the example of an inventory of items with unique Ids. Explain the BOSS accessing methods suitable for this use.

6. Write a program that creates two arrays called array1 and array2, both containing element $x = 5/6$. 'Boss' array1 to file test.1a and array2 to file test.1b, and then boss the two objects back in, storing them in variables array3 and array4. The original arrays array1 and array2 shared the same element x , in other words, there were originally three objects – array1, array2, and x . Arrays array3 and array4, on the other hand, each have their own copy of $5/6$ corresponding to four objects altogether. The objects bossed back are thus an imprecise representation of the original objects.
7. Repeat the previous exercise but write the two arrays to the same BOSS file. What do you get when you read the two arrays back? Compare with the previous exercise and state a conclusion about storing networks of inter-related objects.

10.11 Other ways of storing objects

As you know, parts of the class library can be stored using *file out* and restored using *file in*. The file out procedure saves the source code and adds a few extra characters to separate, for example, one method from another. *File in* uses these extra characters and the compiler to recompile the code and save it back in the library. As a simple example, the file out of the following method in protocol accessing

```
firstName: aString  
    firstName := aString
```

is the following ASCII file:

```
'From VisualWorks(R), Release 2.5 of September 26, 1995 on July 11, 1997 at 12:56:20 am'
```

```
!Name methodsFor: 'accessing'
```

```
firstName: aString  
    firstName := aString! !
```

As we have seen in the previous section, classes can also be stored by BOSS but their restoration requires BOSS classes rather than the compiler.

Classes and their instances can also be stored and restored by methods *storeOn:* and *readFrom:*, both defined in class *Object* and redefined in several classes at lower levels of the class hierarchy. This approach is independent of BOSS but much less efficient and limited, and it is never used in Smalltalk applications. We introduce it only because its implementation is an interesting example of the use of polymorphism and delegation, and because it is the basis of the automatic saving of changes in your library code.

Message *storeOn: aStream* constructs an ASCII string describing the receiver and adds it to the specified stream. Message *readFrom: aStream* then reconstructs the original object from it as in

```
| dictionary stream |  
dictionary := Dictionary new.  
dictionary add: 'Saleem' -> 'Khan'; add: 'Ke' -> 'Qiu'.  
stream := WriteStream on: (String new: 20).  
"Store the Dictionary object in the stream using storeOn:."  
dictionary storeOn: stream.  
"Produces stream on '((Dictionary new) add: ('Ke' -> 'Qiu')); add: ('Saleem' -> 'Khan'); yourself'."  
"Create a copy of the original Dictionary object using readFrom:."  
Object readFrom: (ReadStream on: stream contents)
```

as you can see when you execute this fragment with *inspect*.

If the stream in which the string is stored is external, this approach can be used to store an object in a file and reconstruct it but the representation is bulky.

The basic definition of `storeOn:` in `Object` simply generates messages to create a new instance of the receiver and further messages to initialize its variables. The interesting part of the definition is that it asks each component of the receiver to store itself. Typically, this results in the component asking its own components to store themselves, and so on. You can see how this can create problems if the structure is circular. The definition of `storeOn:` is as follows:

storeOn: aStream

"Append to aStream an expression whose evaluation creates an object similar to the receiver. This is appropriate only for smaller simpler objects and it cannot handle arbitrary circular references of objects."

```
aStream nextPut: $(
self class isVariable
    ifTrue: "For instances of classes with indexable elements."
        [aStream nextPutAll: '(', self class name, ' basicNew: ' ; : self basicSize; nextPutAll: ') ' ]
    ifFalse: "For instances of classes with named instance variables"
        [aStream nextPutAll: self class name, ' basicNew'].
"Get values of instance variables of the receiver object being stored and ask them to store themselves using store:."
1 to: self class instSize do:
    [:i | aStream nextPutAll: ' instVarAt: ' ; store: i;
    nextPutAll: ' put: ' ; store: (self instVarAt: i); nextPut: $;].
1 to: self basicSize do: "Get values of indexed variables"
    [:i | aStream nextPutAll: ' basicAt: ' ; store: i; store: (self basicAt: i); nextPut: $;].
aStream nextPutAll: ' yourself'
```

If the object being stored has some special properties, we may be able to store the object more efficiently. As an example, `Array` redefines `storeOn:` to take advantage of literal arrays as follows:

storeOn: aStream

"Use the literal form if possible."

```
self isLiteral
    ifTrue: [aStream nextPut: $#; nextPut: $(
        self do: [:element | storeOn: aStream. space].
        aStream nextPut: $)]
    ifFalse: [super storeOn: aStream] "Use general implementation if not literal."
```

and the definition of `storeOn:` in class `Point` is

storeOn: aStream

```
aStream nextPut: $(;
nextPutAll: self species name;
nextPutAll: ' x: ' ;
store: x;
nextPutAll: ' y: ' ;
store: y;
nextPut: $).
```

Note that the basic definition of `storeOn:` depends on method `store:` which is defined in class `Stream` as follows:

store: anObject

"Have anObject print itself on the receiver for rereading."

```
anObject storeOn: self
```

This interesting definition simply exchanges the receiver and the argument of `storeOn:` (`anObject storeOn: aStream` is equivalent to `aStream store: anObject`) to make the definition of `storeOn:` simpler. Since `store:` both uses and is used by `storeOn:`, the definition of `storeOn:` is recursive - when it stores the values of instance variables of an object, it asks them to store themselves (Figure 10.13).

create some of the code and send `storeOn:`
to components to create the rest

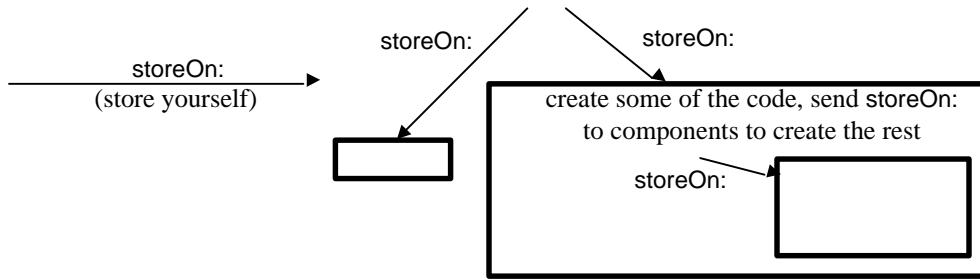


Figure 10.13. The definition of `storeOn:` is recursive.

As an illustration of the operation of this recursive definition, consider using `storeOn:` on a literal array containing string elements: The `storeOn:` method creates the code to create a literal array and asks the string elements to create their own description of how they are stored. As a result, when you *inspect*

```
| stream |
stream := WriteStream on: (String new: 16).
#('ab' 'cd' 'ef') storeOn: stream.
stream
```

you will find that the stream's contents are

```
'#("ab" "cd" "ef" )
```

where the underlined parts were created by the string elements of the array, and the rest by `storeOn:` in Array. When BOSS stores compound objects, it operates the same way.

As a closing note, the simple nature of `storeOn:` does not allow it to handle circular structures – unlike BOSS which does.

Main lessons learned:

- Methods `storeOn:` and `readFrom:` can store and reconstruct any object that does not have circular structure.
- Both `storeOn:` and `readFrom:` are used by the system but applications use either BOSS or a data base system to store objects in files.
- Method `storeOn:` is recursive and delegates the storage of the components of the object being stored to the components themselves.
- Method `storeOn:` cannot handle circular structures.

Exercises

1. What is the difference between `printOn:` and `storeOn:`?
2. Examine and explain the result of executing `storeOn:` on the object created with message Array with: 'abc' with: (Array with: with: 13 \$x with: 5 factorial) with: (Dictionary with: ('key' -> 'value')). Test that `readFrom:` reconstructs the original object.
3. Execute the previous exercise with an external stream and file and open a file editor on the file. Compare the contents of the file with the contents of the equivalent BOSS file.
4. Explain the definition of `readFrom:` in three selected classes.
5. Rewrite the definition of `storeOn:` in Object without using `store:` to appreciate the gain in simplicity.
6. Explain the definitions of `storeOn:` in the following classes: Character, Collection, Date, and Time.

7. How would a Point be stored by the original storeOn: method defined in Object? How is it stored by its special storeOn: method? (Hint: Redefine storeOn: in Point to use super storeOn:.)
8. Find references to storeOn: and readFrom:.
9. Compare the speed and storage requirements of storeOn: and readFrom:, and BOSS, by storing several arrays of increasing size. Plot the results in terms of speed and file size as a function of the size of the arrays.

Conclusion

Sequenceable collections are often accessed linearly - one element after another. When a loop executing identical statements for each element is desired, this is best implemented with enumeration. When access is irregular, for example dispersed over several consecutive statements, streaming (use of streams for accessing) is preferable because it eliminates the need to maintain the current position within the collection. This becomes almost essential when the access is distributed over several methods.

All streams are subclasses of the abstract class Stream and can be divided into three groups: class Random, internal streams, and external streams. In this chapter, we dealt with internal and external streams and their three main forms - read-only, write-only, and read-write streams.

Internal streams are used for streaming over sequenceable collections, mainly strings and ordered collections. Their main uses are for string processing and their advantage is increased clarity of programs, simplification of programming, and sometimes increased execution speed. Internal streams are heavily used by the system and experienced programmers but novice programmers often don't take advantage of them and access collection elements by their index when streaming would be preferable.

External streams are used for accessing files and networks in a byte-by-byte fashion. In VisualWorks, files are implemented as instances of class Filename. Class Filename implements Smalltalk's interface to the platform's file system and executes various file-oriented and disk-oriented operations without explicit cooperation of an external stream. (Some of these operations use an external stream but hide it.) When the operation requires explicit access to the elements of the file, an external stream of the appropriate kind must first be created by sending a stream-creation message to the Filename object. Filename objects themselves are created with a string specifying the name of the file or directory. Class Dialog provides several powerful file-request messages that make obtaining the name of a file easier.

To use external streams and files properly, one must understand that external streams use an intermediate buffer object to hold a working copy of a part of the file or transmitted data. Sending flush to the stream sends the contents of the Smalltalk buffer to the operating system's buffer, commit sends it directly to the disk. The buffer is also flushed or committed whenever it becomes full, and committed when the file is closed by sending the close message to the stream.

The Binary Object Streaming Service (or BOSS) stores and restores objects efficiently and with minimum effort on the part of the programmer. To use BOSS, specify the file, create an appropriate external stream over it, create a BinaryStorageObject over the stream, perform the required operation, and close the BinaryStorageObject object. We have only covered basic storage and retrieval of class instances; more sophisticated uses are described in the User Manual. There are only a few occasions when an application might need to deal with a file directly, such as when you want to read a file containing a digitized picture rather than a Smalltalk object.

Two other ways of storing and restoring objects are the use of a data base system (VisualWorks library does not contain one), and the storeOn: and readFrom: messages. These two messages are heavily used by the system to save changes to the library but not by applications because they are very inefficient.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

BufferedExternalStream, ExternalStream, ExternalReadAppendStream, ExternalReadStream, ExternalReadWriteStream, ExternalWriteStream, **Filename**, InternalStream, PositionableStream, ReadStream, ReadWriteStream, Stream, **TextStream**, WriteStream.

Terms introduced in this chapter

append stream - stream allowing adding elements only at the end

binary object storage - storage of objects in binary form rather than as printable ASCII codes

buffer - area in memory holding data such as a part of a file

commit - save contents of a buffer on the disk

external stream - a stream designed for file or network access

file handle - a binary number used by the operating system to refer to a file

internal stream - accessor of sequenceable collections such as strings and ordered collections

stream - an accessor of linearly arranged data

streaming - linear access of sequentially organized data using a stream

Chapter 11 - Stacks, queues, linked lists, trees, and graphs

Overview

Although the collection classes presented in previous chapters are sufficient for most tasks, several other structures for holding collections of objects are also commonly used. The most important of them are stacks, queues, linked lists, trees, and graphs. This chapter is an introduction to these structures with emphasis on intuitive rather than most efficient implementations. For a more advanced treatment, we recommend one of the many books on data structures.

A stack is a collection whose elements can be accessed only at one end called the top of the stack. The operation adding an element on the top of the stack is called push, the operation removing the top element from the stack is called pop. Implementing stacks in Smalltalk does not require a new class because stack behavior is subsumed by `OrderedCollection`. Our coverage will thus be limited to several examples of the uses of the stack.

A queue is a collection in which elements are added at one end and retrieved at the other. Its familiar real-life example is a line in a bank. Queues do not require a new class because their behavior is also a part of the behavior of `OrderedCollection` and our presentation will thus be limited to examples.

A linked list is a linearly arranged collection of elements that allows insertion and deletion at any place in the sequence. This behavior is necessary in many applications and not easily achieved in the collections presented so far. Smalltalk's class `LinkedList` implements a basic linked list.

A tree is a structure whose graphical representation looks like a family tree: It starts with a root at the top, and branches downward. Typical uses of trees are the representation of the class hierarchy, storing data for fast access, and translation of program code. Computer applications use many kinds of trees but Smalltalk does not contain a general-purpose tree class. We will develop a class implementing the simplest kind of tree - the binary tree.

Graphs can be used to represent concepts such as road maps, house plumbing diagrams, and telephone networks. They consist of nodes and connections between them. Graphs have many different applications but Smalltalk does not have any built-in graph classes because the system does not need them. We will design and implement a graph class and demonstrate a few typical graph operations.

11.1 Stack - an access-at-top-only collection

A stack is usually defined with reference to a stack of cafeteria trays. New objects are added on the top by the push operation, and existing elements can only be removed by the pop operation which removes the top element (Figure 11.1). For obvious reasons, a stack is also called a last-in first-out (LIFO) collection. Stacks are very important in several areas of theoretical Computer Science and in the process of computing.

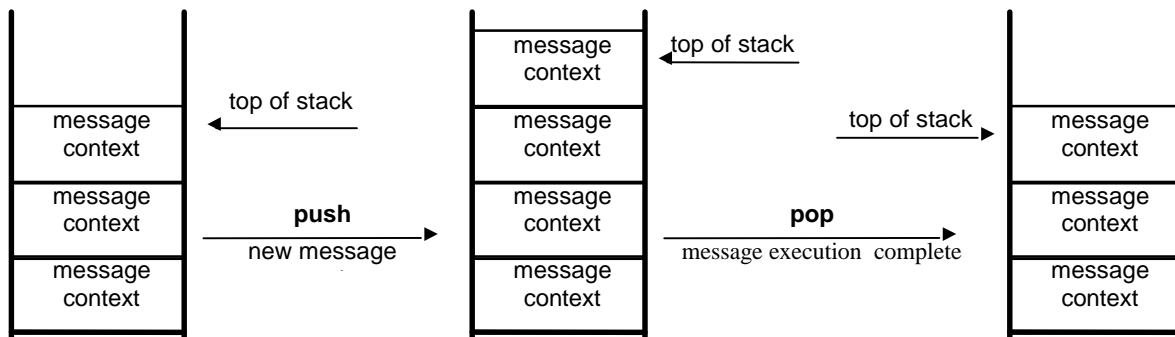


Figure 11.1. Execution of Smalltalk messages is based on a stack of 'contexts'.

In essence, a stack is an ordered collection whose elements can only be accessed at one end. If we treat the start of the collection as the top of the stack, `addFirst:` performs push and `removeFirst` performs pop. Alternatively, we can use the end of the `OrderedCollection` as the top of the stack with `addLast:` for push and `removeLast` for pop. If we need a stack and if we want to restrict ourselves to the stack-like behavior of `OrderedCollection`, there is thus no need to define a new `Stack` class and this is the approach taken in the built-in `VisualWorks` library. From a strict OO point of view, however, this approach is not appropriate because it leaves the simulated stack object open to all behaviors of oo instead of restricting it to the very small behavior of stacks.

In the following, we will restrict our coverage to two examples from the Smalltalk environment and leave an implementation of a `Stack` class as an assignment. Our first example is a behavior that resembles stacks but is not really a stack, the second is a very important use of a stack at the core of Smalltalk implementation.

Example 1: The stack-like behavior of the *paste* operation

The *paste* command in the text editor pop up menu can paste any of the recently cut or copied strings. To do this, press <shift> when selecting *paste*. This opens a dialog (Figure 11.2) displaying the most recent *copy* or *cut* string at the top and the oldest *copy* or *cut* string at the bottom.

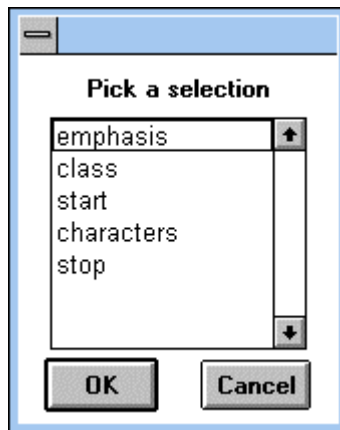


Figure 11.2. <shift> *paste* displays the latest copy/cut strings and allows selection.

Although this behavior is based on the stack principle and demonstrates its main purpose - keeping recent information accessible in the last-in first-out style - the structure is not strictly a stack: For one thing, the definition of the stack in class `ParagraphEditor` restricts its depth to five strings. To implement this restriction, updates of the `OrderedCollection` discard the element at the bottom when the size reaches five elements. Also, before adding a new string, the definition first checks whether the string already is on the top of the stack and if it is, it does not duplicate it. The main difference between the copy buffer and a true stack is that the user can select any string in the buffer and not only the top one.

It is interesting to note that the string buffer is held in a *class* variable of `ParagraphEditor`, making it available to any instance of `ParagraphEditor`. As a result, any of the last five strings copied from *any* text editor can be pasted into *any* text editor.

Although the paste buffer is not a pure implementation of a stack, its behavior is a nice illustration of the usefulness of the concept.

Example 2: Stack as the basis of message execution

When a program sends a message, the context associated with it - the code and the objects that the message needs to execute - are pushed on the top of the context stack. During execution, the Smalltalk *object engine* (the program responsible for managing program execution; also called the *virtual machine*) accesses this information to obtain the objects that it needs and the messages to send. When sending a message, the virtual machine creates a new context for this message, puts it on the stack, and starts its execution. When the message is finished, its context is popped from the stack and execution returns to the sending message (Figure 11.1).

The principle of message execution does not require further elaboration and we will thus dedicate the rest of this section to a brief discussion of contexts, the objects stored on context stacks. The reason for including this subject is its essential importance for understanding Smalltalk operation, and the fact that it illustrates the importance of stacks.

Context and related concepts

To start this discussion, we will first put message execution into the context of transformation of the source code into an executing program.

Before we can execute a method, its source code must be compiled, for example by clicking *accept*. In response to this, the compiler produces an internal representation of the code whose essential part is an instance of `CompiledMethod` (to be discussed shortly). We will now demonstrate how the compiling process works and what it produces.

Consider a class called `Test` and its instance method

```
test1: anInteger
| temp |
temp := anInteger factorial.
^temp
```

defined in protocol `test`. To see how this method is compiled and added to protocol `test` of class `Test` (with the same result as if you clicked *accept*) execute

```
self halt.
Test compile: 'test: anInteger |temp| temp := anInteger factorial. ^temp'
classified: 'test'
```

When you observe the operation in the Debugger, you will find that execution of this expression consists of two main steps: In the first step, the code is compiled, producing a `CompiledMethod` object which is inserted into the method dictionary of class `Test`. In the second step, this object is complemented by the source code and other information. To see the result, *inspect*

Test compiledMethodAt: #test1:

The returned `CompiledMethod` object has several instance variables and the most interesting ones contain the source code of the method and the *bytecodes* stored in instance variable `bytes`. Under `byte codes`, you will find the following:

short CompiledMethod numArgs=1 numTemps=1 frameSize=12

literals: (#factorial)

```
1 <10> push local 0
2 <70> send factorial
3 <4D> store local 1; pop
```

```
4 <11> push local 1
5 <65> return
```

As you can see, a CompiledMethod contains information about the number of arguments of the method, the number of its temporary variables, a list of literals - the messages sent by this method (only #factorial in this case), and a sequence of bytecodes - an internal representation of the source code ready for execution by the stack-based virtual machine¹. Let us now examine the bytecodes in more detail:

```
1 <10> push local 0
2 <70> send factorial
3 <4D> store local 1; pop
4 <11> push local 1
5 <65> return
```

The codes in brackets in the second column are expressed in hexadecimal notation, a shorthand for internal binary representation. They are the translation of the original source code and represent ‘opcodes’ of a fictitious Smalltalk CPU. This CPU does not in reality exist but is emulated by the virtual machine which interprets² the codes and produces the effect described next to the byte code. As an example, hexadecimal code 10 has the effect of pushing the value of the first argument of the message on the stack of intermediate results. We will now illustrate how the “interpreter” executes the byte codes representing the method, assuming the following situation:

test

“Some method. containing test1.”

```
...
Test new test1: 5
...
```

When the virtual machine encounters, for example, the message test1: 20 (its CompiledMethod equivalent produced by the compiler), it puts its context on the context stack (more on this later) and creates an *evaluation stack* to hold the intermediate results of execution (Figure 11.3). It then starts executing the byte codes of the test1: method one after another starting with the code in position 1:

1. Code 10: Push the value of argument 20 (‘local object 0’) on the evaluation stack.
2. Code 70: Send message factorial (‘literal 0’) to the object on the top of the evaluation stack. This finds and executes the CompiledMethod with the byte codes of factorial (not shown), and leaves the result (SmallInteger 720) on the top of evaluation the stack, replacing the original object (SmallInteger 20). Control returns to the test1: method. (This factorial message send is executed in the same way as the test1: message that we are now tracing.)
3. Code 4D: Stores the value on the top of the evaluation stack (the result of 20 factorial) in temp (‘local 1’) and pops the stack, removing the 20 factorial value. This step is equivalent to the assignment part of the assignment statement in the source code.
4. Code 11: Push the temp object (‘local 1’) on the evaluation stack.
5. Code 65: Return to the message that sent test: (in this case message test), pushing the value of temp - the value that test1 was supposed to return - on the top of its evaluation stack.

¹ For more on the virtual machine, see Appendix 8.

² The statement about ‘interpretation’ was strictly true for earlier implementations of Smalltalk but modern implementations translate bytecodes into the machine code of the CPU running the program when the method is first invoked during execution. This process, called *dynamic compilation* or *just in time (JIT) compilation* makes execution more efficient. Once compiled, the machine code is stored in a code cache so that it does not have to be retranslated.

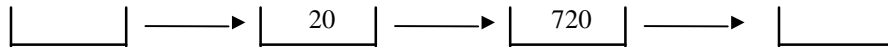


Figure 11.3. Effect of execution of test1: on its evaluation stack. In this case, the execution stack never contains more than one element but other methods may require a deeper stack.

Let us now analyze what information the virtual machine needs to have, to be able to do what we have just described. To execute a message, the virtual machine must know

- the *bytecodes* of the message with information about its arguments and temporary variables
- the *sender* of the message (to be able to transfer the result back to it)
- the *receiver* of the message (required by *self*, *super*, and for access to instance variables)
- an *evaluation stack* for calculations required by byte codes
- the *current position* in the execution of the byte code sequence; this object is referred to as the *program counter* or the PC of the virtual machine.

The object containing all this information is called a *context* and it is an instance of class *MethodContext*.

For a concrete illustration of these concepts, create the following method

test2

```
| context temp |  
  context := thisContext.    "thisContext is a special variable like self and super. It returns the  
                             currently active context."  
  temp := 3 + 7 * 5 factorial.  
  ^temp
```

in class *Test* and execute

```
self halt.  
Test new test2
```

In the Debugger, execute the message step-by-step and observe the changing value of PC in the context variable. Observe also the stack and the stack pointer.

Now that you understand the use of a context in execution, let's look at an example of execution in a broader setting. First, create the following three methods in class *Test*:

test3

```
| context |  
  context := thisContext.  
  self test4.  
  ^self
```

test4

```
| context |  
  context := thisContext.  
  self test5.  
  ^self
```

test5

```
| context |  
  context := thisContext.  
  ^self
```

Next, execute the following expression and observe how control passes from one method to another, how their contexts are pushed on top of one another in the context stack, and how they are popped when execution of the method ends:

self halt. Test new test3

These examples illustrate the critical importance of the stack in Smalltalk: At all times, Smalltalk operation directly depends on two stacks - one for contexts (representing message code), the other for intermediate results of the currently executing message. Whereas the context stack is shared by all contexts, each invoked message has its own working stack associated with its context.

Main lessons learned:

- A stack is a last-in first-out (LIFO) structure. Elements are added on the top using the push operation and removed using the pop operation.
- The behavior of a stack is included in the behavior of `OrderedCollection` and there is no need for a `Stack` class.
- Execution of Smalltalk messages depends on a stack of context objects, each of them carrying all information about a message, its receiver and sender, its arguments and local variables, and current state of execution.
- Translation of the source code of a method into a form executable on the virtual machine is expressed in bytecodes.
- A part of the context of each executing method is an evaluation stack holding intermediate results.

Exercises

1. Examine the nature of the copy buffer; in particular, check whether its elements must be strings. If not, can you think of some additional uses of the copy buffer?
2. When implementing a stack, is it better to use the end or the start of an `OrderedCollection` as the top?
3. Implement class `Stack` with only the essential stack behavior.
4. While debugging, you might want to print the names of selected messages whenever they are sent. You could, of course, use the `show:` message with the name of the selector explicitly spelled out as in Transcript show: 'Executing with:with:with:' but this is somewhat awkward. A neater solution is to extract the name of the method from the context. Implement this approach via a new method called `printMethodName:`.
5. Browse context-related classes and write a short description.

11.2 Context Stack and Exceptions

As another illustration of the use of stacks, we will now implement *Tester*, a tool to help automate the testing of classes. Testing is a very serious concern in software development because all code must be carefully tested before it is delivered to customers. Because testing generally requires verifying program behavior with many test conditions, the process may be very time consuming and expensive. To minimize the effort and time required for this part of program development, software tools have been developed to automate the process as much as possible. In Smalltalk, these test programs often assume that a class under test contains special testing methods in a special protocol, search these methods out, and execute them. We will develop a very simple version of such a program.

Our test tool (to be implemented by class `Tester`) will allow the user to select the class to be tested, locate all the test methods defined in the class, execute them, and write a report to the Transcript. The user interface will be as in Figure 11.4. For each of the executed methods, `Tester` will print the name of the method followed by the result (*success* or *failure*) in the Transcript. If a test method fails, `Tester` also prints

a message to the Transcript. All test methods of a class are assumed to be in class protocol testing, and each test must return true if it succeeds and false if it fails.

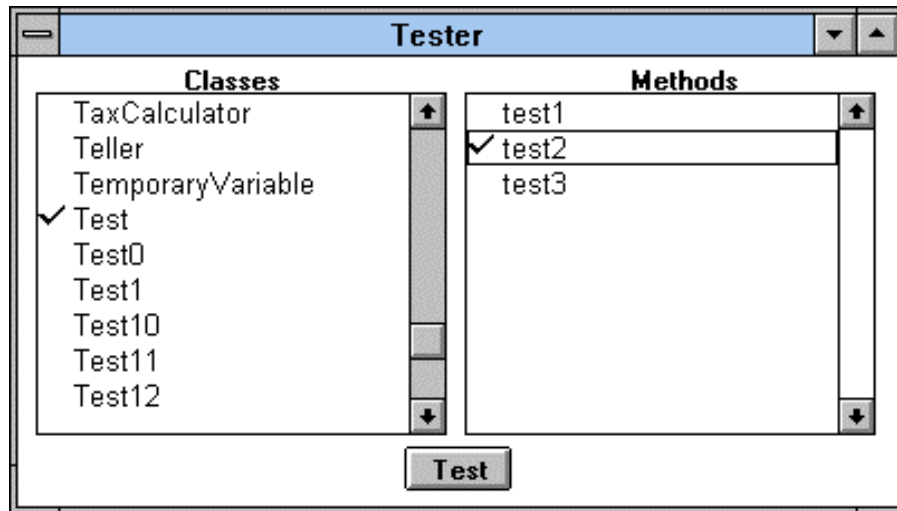


Figure 11.4. Desired user interface for Tester.

The required functionality is described by the following scenarios:

Scenario 1: User selects a class and test methods and successfully runs the test

Conversation:

1. User clicks class to be tested.
2. System displays the list of all methods defined in its class protocol testing.
3. User selects methods to be executed and clicks *Test*.
4. System executes the selected methods and prints report in Transcript.

Scenario 2: User selects a class and test methods, one of the methods fails during execution

Conversation:

1. User clicks class.
2. System displays list of corresponding test methods.
3. User selects methods to be executed and clicks *Test*.
4. System starts executing the selected methods and one of them fails as 'not understood'.
5. System displays appropriate note in the Transcript and executes the remaining methods.

Solution: Scenario 1 can be implemented rather easily but Scenario 2 introduces a problem that we have not yet encountered: The Tester must be capable of completing execution even if a method fails to execute - technically speaking, even when an exception occurs. In the following, we will show how to deal with this problem without explaining how the mechanism works. (We will also skip the user interface of Tester and leave it as an exercise.) In the next section, we will explain the principle of exception handling in VisualWorks and show that it relies on the context stack.

Preliminary considerations. To implement the given scenarios, Tester must know how to

1. obtain a list of class names
2. obtain a list of methods in class protocol testing
3. execute a method and recover if it raises an exception.

As we know, class names are returned by Smalltalk `classNames`. To obtain information about classes, we already learned to use category `Kernel - Classes` and this is indeed where we find an answer to the second question (see also Appendix 3): Instance variable `organization` in `ClassDescription` contains an instance of `ClassOrganizer` that holds information about a class's categories and method names. Sending `listAtCategoryNamed: aSymbol` to this object returns the names of all methods in the specified protocol. As an example, the following expression returns the names of the selectors of all *instance* methods defined in protocol `#update` in class `Object`:

```
Object organization listAtCategoryNamed: #updating
```

and

```
Test class organization listAtCategoryNamed: #testing
```

returns all *class* methods under protocol `#testing` in class `Test`.

Finally, the third requirement. To execute a message 'safely' and recover when an exception occurs, we must use the `handle:do:` message with appropriate arguments. We will explain the details later but for now, suffice it to say that Smalltalk contains a number of predefined 'signal' objects that correspond to various error conditions such as 'division by zero', 'index out of bounds', and 'message not understood', and when such a condition occurs, the signal object can be used to trigger exception-recovery behavior. As an example, the first of the following statements will attempt to execute `3/0`, intercept the attempt to divide by zero, write 'Division by zero' to the Transcript, and continue with the next statement instead of opening an Exception window:

```
ArithmeticValue divisionByZeroSignal  
  handle: [:exception| Transcript cr; show: 'Division by zero'.  
           exception return]  
  do: [ 3 / 0].  
Transcript cr; show: 'Look ma, no exception window'
```

As you can see, the block argument of the `do:` part of the `handle:do:` message contains the operation that we would like to execute, and the block argument of the `handle:` keyword specifies what to do if the `do:` block fails. The `handle:` block has one argument - the Exception object. In our block, we sent message `return` to this argument to request that if the exception occurs, execution should return to the original code (our test program) and continue. Class `Exception` provides various other behaviors explained in the User Guide.

Let's now return to our problem. Assuming that the only possible cause of failure is 'message not understood', the following method will execute each test method, print *success* or *failure* if the method executes (and returns true or false), and print a message to the Transcript if the 'message not understood' exception occurs:

testClass: classToTest methods: methodsToRun

```
"Execute specified test methods and print report to Transcript."  
methodsToRun isEmpty ifTrue: [Dialog warn: 'No method selected'].  
Transcript clear; show: 'Results of test on class ', classToTest name; cr.  
methodsToRun  
  do: [:method |  
    Transcript cr; show: method asString; tab.  
    Object messageNotUnderstoodSignal  
      handle: [:exception |  
        Transcript show: 'Message not understood'.  
        exception return]  
    do: [Transcript show: ((classToTest perform: method)  
                          ifTrue: ['success']  
                          ifFalse: ['failure'])]]
```

We assume that our test methods don't have any parameters but this restriction could be easily removed

To test whether the method works, we created a class called `Test` and defined the following three methods in its class protocol testing:

test1

```
"Should cause an exception."  
3 open.  
^true
```

test2

```
"Should execute and return true."  
^3 factorial = 6
```

test3

```
"Should execute and return false."  
^3 squared = 10
```

Since we don't have the proper user interface, we tested our code by executing

```
Tester new testClass: Test methods: #(#test1 #test2 #test3)
```

which produced the following expected result:

Results of test on class `Test`

```
test1    Message not understood  
test2    success  
test3    failure
```

Watch out and make sure that you put your `Test` test methods in the correct class protocol, otherwise you will get only 'message not understood' reports. We leave the full implementation of `Tester` with the prescribed user interface as an exercise.

Main lessons learned:

- VisualWorks Smalltalk has a built-in mechanism for dealing with exceptional situations. It allows the programmer to anticipate exceptional behaviors and deal with them programatically, preventing the program from raising an exception.
- Exception handling depends on instances of class `Signal`.

Exercises

1. Implement and test the `Tester`.
2. Generalize `Tester` to handle test methods with any number of arguments.

11.3 More about exceptions

To explain the internal operation of exceptions, we will now take our example from the previous section and trace its execution. The sequence is lengthy and we suggest that you read and execute it, read the summary, and reread the trace one more time. Our test code is as follows:

```
self halt.  
ArithmeticValue divisionByZeroSignal  
  handle: [:exception| Transcript cr; show: 'Division by zero'.  
           exception return]  
  do: [ 3 / 0].  
Transcript cr; show: 'Look ma, no exception window'
```

and the main events that occur in its execution are as follows (Figure 11.5):

Expression `ArithmeticValue divisionByZeroSignal` returns the `DivisionByZeroSignal` object. This `Signal` object then executes `handle:do:` which is defined as follows:

handle: handlerBlock do: doBlock

```
"Execute doBlock. If an exception occurs whose Signal is included in the receiver, execute handlerBlock."  
<exception: #handle>  
^doBlock value
```

The message first evaluates the `doBlock` which in our example invokes the `division` message `3/0`. As this message executes, it eventually reaches the following method in class `Fraction`:

reducedNumerator: numInteger denominator: denInteger

```
"Answer a new Fraction numInteger/denInteger."  
| gcd denominator numerator |  
denominator := denInteger truncated abs.  
denominator isZero  
  ifTrue: [^self raise: #divisionByZeroSignal  
                      receiver: numInteger  
                      selector: #/  
                      arg: denInteger  
                      errorString: 'Can"t create a Fraction with a zero denominator'].  
etc.
```

If the denominator argument is 0, the method sends `raise:receiver:selector:arg:errorString:` defined in the `ArithmeticValue` superclass of `Fraction` as follows:

raise: signalName receiver: anObject selector: aSymbol arg: anArg errorString: aMessage

```
^(self perform: signalName) raiseRequestWith: (MessageSend receiver: anObject  
  selector: aSymbol argument: anArg errorString: aMessage
```

This message essentially asks the appropriate `Signal` to 'raise an exception request'. The part

```
self perform: signalName
```

returns `DivisionByZeroSignal`, and

```
MessageSend receiver: anObject selector: aSymbol argument: anArg
```

returns a `MessageSend`. This is an interesting object and we will digress briefly to explain it. Its *print it* execution in the debugger produces

a MessageSend with receiver: 3, selector: #/ and arguments: #(0)

which shows that this object knows the receiver, the selector, and the arguments of a message, and is an instance of class MessageSend, a subclass of Message. The comment of Message is as follows:

Class Message represents a selector and its argument values. Generally, the system does not use instances of Message. However, when a message is not understood by its receiver, the interpreter will make up a Message (to capture the information involved in an actual message transmission) and send it as an argument with the message doesNotUnderstand:.

In other words, instances of Message are not used to execute messages (messages are compiled from byte codes into machine code and executed by the CPU). But when a message is not understood, the system creates a Message, passes it up the inheritance chain from the original receiver to Object to produce an appropriate doesNotUnderstand: message. This explains the secret of doesNotUnderstand:.

Class MessageSend, which is what we are dealing with here, adds several new behaviors and information about the sender. Its class comment is as follows:

A MessageSend represents a specific invocation of a Message. It is essentially a message send represented as an object, and supports protocol for *evaluating* that send.

After this digression, let's return to our original problem. All arguments of raiseRequestWith:errorString: are now available and Signal executes it. This method is defined as follows:

raiseRequestWith: parameter errorString: aString

"Raise the receiver, that is, create an exception on the receiver and have it search the execution stack for a handler that accepts the receiver, then evaluate the handler's exception block. The exception block may choose to proceed if this message is sent. The exception will answer the first argument when asked for its parameter, and will use aString as its error string"

```
^self newException
    parameter: parameter;
    errorString: aString;
    originator: thisContext sender homeReceiver;
    raiseRequest
```

The first thing that happens is self newException according to

newException

"Answer an appropriate new Exception object. Subclasses may wish to override this."

```
^Exception new signal: self
```

This returns a new Exception object whose main function is to know which kind of Signal is associated with it. In our case, this is the DivisionByZeroSignal signal. As the next step, raiseRequestWith:errorString: obtains some additional information and raiseRequest raises the exception, triggering a sequence of events to find the handler code. To do this, raiseRequest searches the context stack from the top down until it finds the context in which everything started, in our case the unboundMethod

```
self halt.
ArithmeticValue divisionByZeroSignal
    handle: [:exception| Transcript cr; show: 'Division by zero'.
            exception return]
    do: [ 3 / 0].
Transcript cr; show: 'Look ma, no exception window'
```

The highlighted handle: block is now evaluated, displays 'Division by zero' in the Transcript, and proceeds to exception return. This message 'unwinds' the context stack (the whole context stack is still there, the exception only accessed and executed the handler block), removing all contexts down to the

originating context and terminating their execution. Our unboundMethod context is now on the top of the stack and execution continues at the point where we left off in our test program.

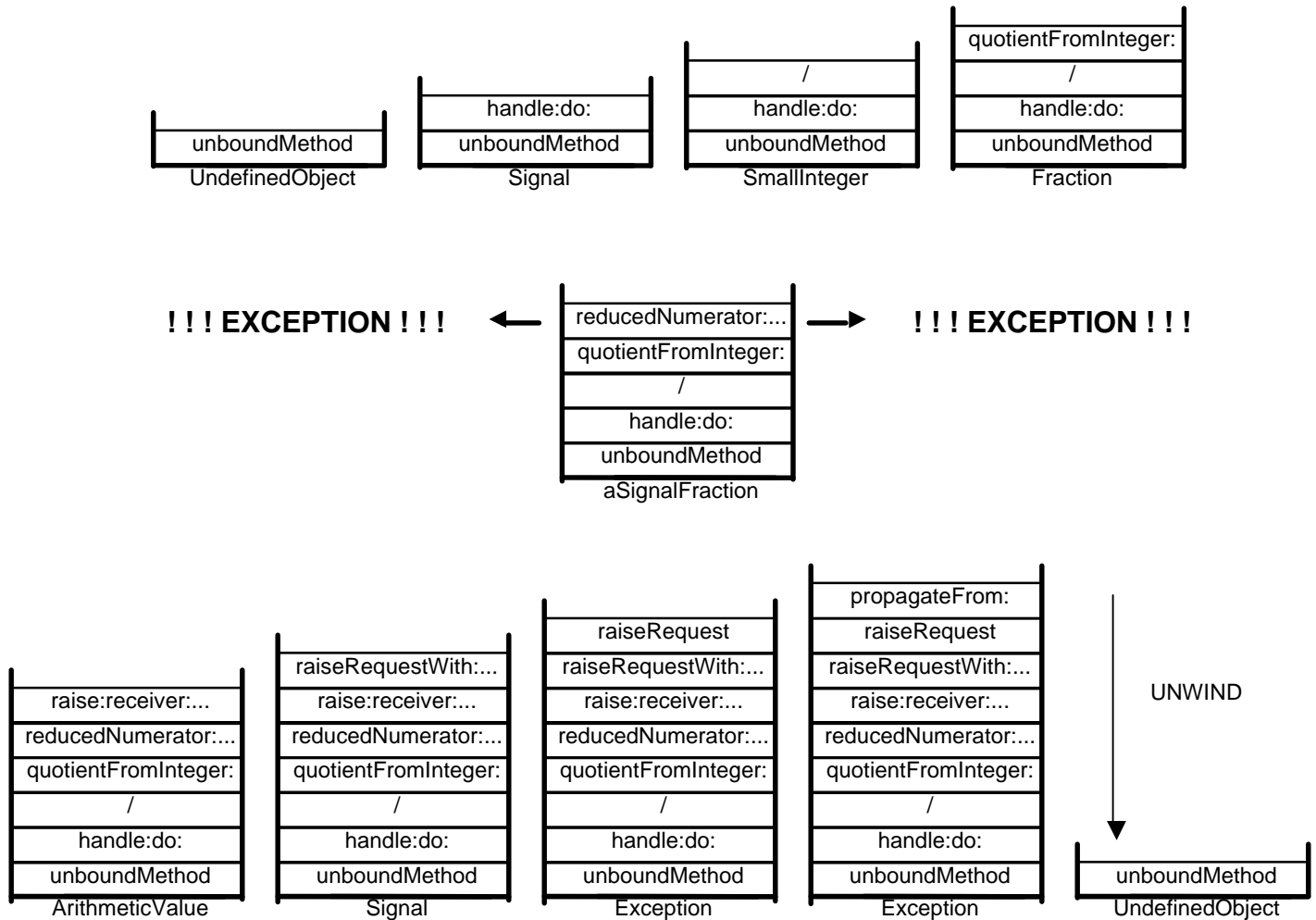


Figure 11.5. Behavior of the context stack in the execution of the test example. The class in which the currently active message is defined is shown below the context stack. Occurrence of exception is indicated.

Let's now summarize what we found: To execute a block of statements safely, let a Signal execute AS IN

```
aSignal handle: blockThatHandlesException do: blockToBeExecutedWhereExceptionCouldOccur
```

The sequence of events is as follows:

aSignal evaluates the do: block. If the block evaluates normally, execution continues to the next message. If a message in the do: block raises an exception addressed to aSignal, aSignal executes the following exception handling mechanism:

- aSignal creates an Exception object.
- The Exception object searches the context stack looking from top downward for the context of the Signal that raised the exception and executes its handler block – the argument of handle:.

c. In our case, the handler sends `return` and in response to this, the `Exception` object unwinds all contexts from the top of the context stack down to and including the `handle:do:` context and execution continues.

We have only scratched the surface of exception handling and we encourage you to explore it further. Exception handling is very important because many situations require catching illegal or special situations and handling them. Some of the examples are attempts to execute a message that the receiver does not understand, sending a mathematical method with inappropriate arguments, accessing an array with an index out of bounds, and failed file access. The principle of implementation of exceptions again shows the critical importance of stacks in Smalltalk's operation.

In closing, note that numerous signals and other ways of handling exceptions are predefined. The following classes contain the most useful signals: `ArithmeticValue`, `BinaryStorage`, `ByteCodeStream`, `ByteEncodedStream`, `ClassBuilder`, `CodeStream`, `ColorValue`, `CompiledCode`, `Context`, `GraphicalContext`, `KeyboardEvent`, `Metaclass`, `Object`, `ObjectMemory`, `OSErrorHandler`, `Palette`, `ParagraphEditor`, and `Process`. The programmer can also define new signals to intercept any desired conditions.

Main lessons learned:

- Signal delegates the handling of exceptions to an instance of `Exception`.
- Exception handling depends on the context stack.
- A number of `Signal` objects are built into the library and users can define their own as well.

Exercises

1. Assume that two other exceptions that might occur in our test methods are division by zero and subscript out of bounds. Modify `Tester` to intercept these exceptions. (Hint: Use class `HandlerList`.)
2. Modify `Tester` to catch *any* error condition. (Hint: Examine the `Object` protocol.)
3. Explain the role of the `MessageSend` object in exception handling.
4. `MessageSend` objects can be evaluated as in (`MessageSend receiver: 3 selector: #factorial`) value. Execute `3 + 4` and `3 between: 5 and: 17` using this technique.
5. What happens when you remove exception return from the `handle: block`?
6. What other messages can be sent to the `Exception` object in the `handle: block` and what is their effect?
7. Trace the operation of `doesNotUnderstand:` by executing `$s factorial`. Write a short description.

11.4. Queues

A queue is a collection of linearly ordered elements in which elements are added at one end and retrieved at the other end (Figure 11.6). As in a queue in a bank, the first item entering the queue is also the first to be retrieved and removed from the queue and this is why a queue is also called a first-in-first-out (FIFO) structure.

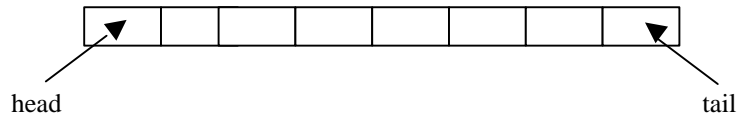


Figure 11.6. In a queue, elements are added at the tail and removed at the head.

In spite of its simplicity, the queue is a very important concept with many applications in simulation of real life events such as lines of customers at a cash register or cars waiting at an intersection, and in programming (such as printer jobs waiting to be processed. Many Smalltalk applications use a queue but instead of implementing it as a new class, they use an `OrderedCollection` because it performs all the required functions³. Since the concept is so simple, we will limit ourselves to an illustration of its use on two examples, one in this section and one in the next.

Simulating a bank queue

At the beginning of the design of a new bank outlet, the designer needs to know how many tellers to provide to satisfy the expected number of customers, and the management will need how to staff the tellers to guarantee satisfactory but economical handling of customers. In our example, the management wants to simulate the following situation in order to evaluate the need for tellers, the cost of operation, and the number of customers that can be handled.

Problem: A bank has a certain fixed number of teller stations (Figure 11.7). Customers arrive at unpredictable random times and queue up for service. There is a single queue and when a teller becomes available, the customer at the head of the queue goes to this teller. Each customer has an unpredictable number of transactions and takes an unpredictable amount of time to process. Our task is to develop and test a program to simulate this situation - but not to use it to make managerial decisions.

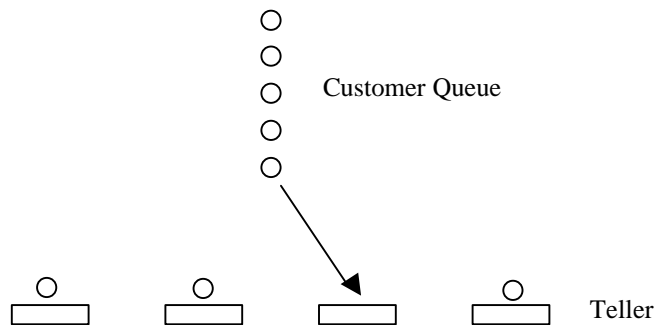


Figure 11.7. Bank layout.

Preliminary considerations: The task has several components. We must

- decide how to model the random arrivals of customers and their random servicing times,
- identify the external parameters that characterize the problem,
- identify expected results
- identify, design, and implement the classes involved in the simulation,
- decide on the user interface.

³ Just as with stacks, a cleaner implementation would be a `Queue` class with behaviors limited to those required by a queue.

Modeling randomness

We will assume that the unpredictable parts of the behavior of the problem can be described by a probabilistic model. In other words, we will assume that we can formulate a mathematical function describing the probability that a new customer will join the queue within the next n minutes, and we will use a random number generator to generate customers according to this formula. We will use the same principle to determine how much time each individual customer spends at the teller.

Practitioners of simulation use a variety of probabilistic models in their simulations, all of them based on specific mathematical assumptions. Matching the situation at hand with the proper set of assumptions requires some knowledge of probability and an understanding of the domain that is being simulated and we will restrict ourselves to the simplest model – we will assume that the distribution of arrival times is *uniform*. In other words, we will assume that there is a certain minimum and maximum inter-arrival time, and that any time between these two limits is equally likely to occur. The advantages of this model are that it is simple and that generation of random numbers is already implemented by class `Random`, its disadvantage is that it does not describe a bank queue well. We will leave a more realistic model as an exercise.

External parameters

In order to make simulation possible, we must identify the parameters that must be specified to start a new simulation. From the specification of the problem and from our discussion of modeling of randomness, it is clear that we need the following parameters:

- Total number of tellers.
- Minimum inter-arrival time.
- Maximum inter-arrival time.
- Minimum expected time required to service a customer.
- Maximum expected time required to service a customer.
- Desired duration of simulation in terms of time or number of customers.

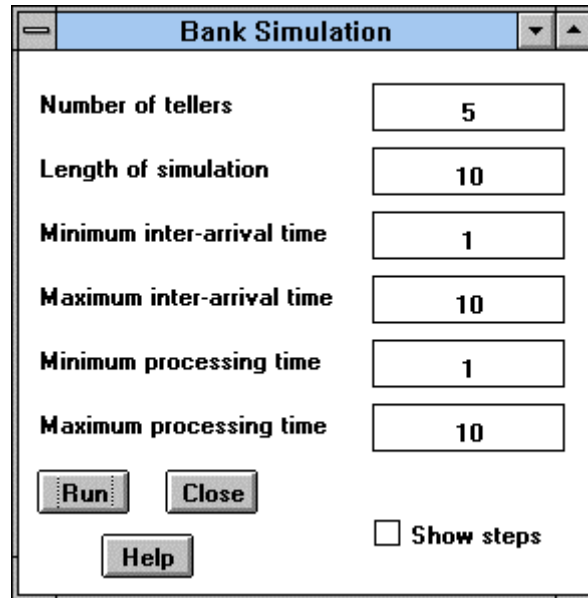
All time-related parameters are expressed in fictitious time units.

Expected results

We will restrict ourselves to creating a log listing customer arrival times, customer departure times, and the average length of the queue calculated over the whole simulation..

Desired user interface

The user interface must make it possible to enter any combination of input parameters and run a simulation. The input of data will be as in Figure 11.8 and the results will be printed in the Transcript ordered along the fictitious time axis.



The image shows a graphical user interface window titled "Bank Simulation". It contains several input fields for simulation parameters, each with a label and a text box containing a value. The parameters are: "Number of tellers" (5), "Length of simulation" (10), "Minimum inter-arrival time" (1), "Maximum inter-arrival time" (10), "Minimum processing time" (1), and "Maximum processing time" (10). Below these fields are three buttons: "Run", "Close", and "Help". To the right of the "Help" button is a checkbox labeled "Show steps", which is currently unchecked.

Parameter	Value
Number of tellers	5
Length of simulation	10
Minimum inter-arrival time	1
Maximum inter-arrival time	10
Minimum processing time	1
Maximum processing time	10

Buttons: Run, Close, Help

Checkbox: ☐ Show steps

Figure 11.8. Desired user interface. The *Show steps* check box determines whether all customer transfers are output to the Transcript or not.

Class exploration

The objects immediately identifiable from the specification are customer objects (class *Customer*), tellers (class *Teller*), and the queue (class *Queue*). We also need an object to generate new customers and add them to the queue (class *CustomerProducer*). With this basic set of objects, let us explore how the simulation might proceed:

- The *CustomerProducer* generates a new customer and adds it to the queue. It then creates a random inter-arrival time t and another customer who will enter the queue at time t .
- The customer in the queue is allocated to the first available teller. (This is not quite fair because the first teller will be getting more work than others and we should allocate customers to tellers randomly. In our problem, such considerations are irrelevant because they don't change anything on the outcome.)
- The teller starts processing the customer and releases him or her after the amount of time required for processing by this particular customer object. This parameter is generated by *CustomerProducer*.
- From this point, execution continues along the following lines until the total length of simulation is completed:
 - If one or more tellers are available and a customer is waiting, the customer is sent to the first available teller.
 - When the inter-arrival time expires, the *CustomerProducer* generates a new customer object and adds it to the end of the queue. It also generates a new inter-arrival time to be used to generate a new customer.
 - Processing of customers by tellers is as explained above.

This algorithm suggests that the simulation is driven by time – the ticks of a fictitious clock determine when new customers are produced and when they are released by teller. We thus need a time managing object (class *SimulationManager*) whose responsibility will be to notify the *CustomerProducer*, *Teller*, and *Queue* objects when a unit of time expired. These objects will be responsible for taking an appropriate action. We also need to take care of the results. We will have the *Queue* and *Teller* objects report the appropriate information and the *Queue* object will be responsible for calculating the average

queue length. The output will, of course, be left to the application model (class BankSimulation) to display. This way, if we want to change the user interface (output of results) or make our simulation a part of a larger scheme, the domain objects can stay the same and only the application model must be changed.

Preliminary design of classes and their responsibilities

We have, so far, identified need for the following classes:

- BankSimulation. In charge of user interface: Input of parameters, start of simulation, output of results.
- Customer. Knows when it entered the queue and how much processing time it will require from a teller.
- CustomerProducer. Generates Customer objects with random values of processing time, keeps information about inter-arrival time and uses it to produce another customer when the time arrives.
- Queue. Knows its customers, can accept new Customer objects, knows how to check whether a Teller is available, knows how to send a Customer to a Teller. Calculates queue statistics and notifies BankSimulation when a customer is added.
- SimulationManager. Starts, runs, and ends simulation. Collects results at the end and notifies BankSimulation. Issues timing information to CustomerProducer, Queue, and Teller objects.
- Teller. Knows how to accept and process a customer, knows whether it has a Customer. Notifies BankSimulation when a customer arrives or is released.

Is this set of classes complete? To find out, we will re-execute our informal scenario dividing it into three phases: simulation start up, body (repeated over and over until the end of simulation), and end of simulation. The following is a rather detailed description with comments on the feasibility of individual steps within the existing class descriptions:

Phase	Description of step.	Comment
Start up	BankSimulation gets valid parameters and starts SimulationManager.	OK
	SimulationManager initializes time and asks CustomerProducer, Queue, and Teller to initialize themselves.	OK
	CustomerProducer generates a Customer with <i>random processing time</i> and adds it to the Queue.	<i>How?</i>
	Queue tells BankSimulation that it received a Customer.	OK
	BankSimulation outputs the event to Transcript.	OK
	Queue sends Customer to the first available Teller.	OK
	Teller tells BankSimulation that it received a Customer.	OK
	BankSimulation outputs the event to Transcript.	OK
	CustomerProducer generates another Customer with <i>random processing time</i> and assigns it a <i>random waiting time</i> (customer 'waits in front of the bank'). It will now wait to release the customer to Queue.	<i>How?</i>
Body	SimulationManager increments time and checks for end of simulation. If not end, it informs CustomerProducer, Queue, and Teller about time change.	OK
	CustomerProducer checks whether to release a Customer. If so, it sends Customer to Queue which updates and notifies BankSimulation. BankSimulation outputs the event to Transcript. CustomerProducer creates new Customer and inter-arrival (waiting) time.	OK
	Each Teller checks whether to release its Customer. If so, it releases Customer and notifies BankSimulation which outputs the event to Transcript.	OK
	Queue checks whether it has a waiting Customer. If so, it checks whether a Teller is available.	OK

	If so, it sends Customer to the first available Teller, Teller calculates time to release this Customer, notifies BankSimulation of arrival, BankSimulation outputs the event to Transcript. Repeated until either no more customers in queue or all tellers busy.	
End of simulation	SimulationManager calculates average wait time and sends this information and total number of customers to BankSimulation.	OK
	BankSimulation outputs the result to Transcript.	OK

Our conclusions from this analysis are as follows:

- We need a new class to generate random numbers uniformly distributed between two positive integers (we will call this class RandomInteger).
- We have reassigned the responsibility for reporting events and results to the objects that are involved in the events. This has the following consequences:
 - Queue needs to know about BankSimulation so that it can communicate results.
 - Teller needs to know about BankSimulation so that it can communicate results.
 - Since Queue is created by SimulationManager, SimulationManager must know about BankSimulation.

The last point is worth closer examination. In our problem, we have two choices to implement the generation of the report. One is to have BankSimulation poll all objects that have something to report, the other is to leave it to the components to notify BankSimulation (Figure 11.9). The first solution has several disadvantages: One is that whenever we change the simulation by adding new types of objects, we must modify the corresponding part of BankSimulation. Another disadvantage is that as the number of objects that have something to report gets larger, the complexity of methods in BankSimulation that perform the gathering of reports also increases. Eventually, BankSimulation will become much too large and its role in the application too predominant. Finally, polling is based on the assumption that the polled object may or may not have something to report. If it does not, polling wastes time. The second approach - leaving the responsibility to report an event to the object that experienced the event (*event-driven* design) can be more efficient. We selected the event-driven approach and leave the polling approach as an exercise.

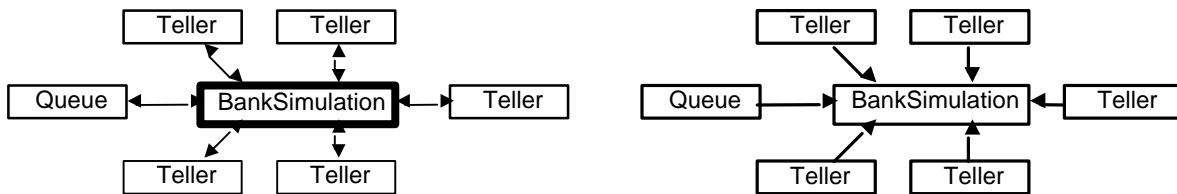


Figure 11.9. Centralized control (left) results in unbalanced distribution of intelligence. Distributed intelligence (right) is generally preferred. The 'intelligence' of an object is indicated by the thickness of the line around it.

Final design

We are now ready to write detailed descriptions of all required classes. We will use the term *reporter* to refer to the BankSimulation object because is responsible for reporting results:

BankSimulation: In charge of user interface - input of parameters, start of simulation, output of results.

Attributes: simulationManager, simulation parameters (number of tellers, minimum and maximum customer processing and arrival times), other aspect variables.

Responsibilities and collaborators:

- Output - implemented by
 - displayEvent: aString. Writes aString to Transcript followed by cr.
- User interface action buttons
 - run - starts simulation
 - close - close application
 - help - display help window

Customer: Represents customer with timing information.

Attributes: processingTime (time to spend at teller station), timeToQueue (time at which Customer entered Queue - for calculation of time spent in bank).

Responsibilities and collaborators:

- Creation - implemented by
 - newWithProcessingTime: anInteger. Creates new Customer. Collaborates with Customer.

CustomerProducer: Generates Customer objects with random values of processing time, keeps information about inter-arrival time and uses it to send Customer to Queue when the time expires and produces another Customer.

Attributes: customer (customer waiting to be released to the queue), releaseTime (when to release current Customer), gapGenerator (random number generator calculating inter-arrival times), processingGenerator (random number generator calculating Customer processing times).

Responsibilities and collaborators:

- Creation - implemented by
 - newWithGapGenerator: aRandomInteger withGapGenerator: gapGenerator withProcessingGenerator: processingGenerator . Collaborates with RandomInteger.
- Updating - implemented by
 - updateTime. Updates time, sends Customer to Queue and creates a new one if appropriate. Collaborates with Queue.

Queue: Knows its customers, can accept new Customer objects, knows how to check whether a Teller is available, knows how to send a Customer to a Teller. Calculates queue statistics and notifies BankSimulation when a customer is added.

Attributes: customers, reporter (reference to BankSimulation), tellers, time (fictitious simulation time)

Responsibilities and collaborators:

- Creation - implemented by
 - numberOfTellers: anInteger reporter: aBankSimulation. Also creates the required number of Teller objects. Collaborates with Teller.
- Processing - implemented by
 - updateTime. Checks whether it has Customer; if so, checks if there is an available Teller; if so, sends Customer to it. Repeated until Queue is empty or no more tellers available. Collaborates with RandomInteger, Teller.
 - addCustomer: aCustomer. Add Customer at end of queue and report it to BankSimulator. Collaborates with BankSimulator.

RandomInteger: Generates random integer numbers within prescribed range.

Attributes: randomGenerator (reference to Random), lowerLimit, upperLimit

Responsibilities and collaborators:

- Creation - implemented by

- lowerLimit: anInteger upperLimit: anInteger.
- Accessing - implemented by
 - next. Returns random integer.

SimulationManager: Starts, runs, and ends simulation. Collects results at the end and notifies BankSimulation. Issues timing information to CustomerProducer, Queue, and Teller objects.

Attributes: customer, lengthOfSimulation, producer, queue, reporter, totalCustomers (total number of customers sent to Queue), totalTimeInBank (sum of times spent in bank by all customers), simulation parameters.

Responsibilities and collaborators:

- Creation - implemented by
 - tellers: anInteger reporter: aBankSimulator simulationLength: anInteger, minProcessing: anInteger maxProcessing: anInteger minArrival: anInteger maxArrival: anInteger. Collaborates with Queue.
- Simulation - implemented by
 - run. Starts time, creates CustomerProducer, creates Queue, creates one object and puts it in Queue, asks CustomerProducer to produce another Customer and hold it until release. Repeats time update notifying Queue and CustomerProducer, moving Customer to queue when released by CustomerProducer. Tells reporter about number of processed customers and average time in bank. Collaborates with Queue, CustomerProducer.

Teller: Knows how to accept and process a customer, knows whether it has a Customer. Notifies BankSimulation when a Customer arrives or is released.

Attributes: customer, customerReleaseTime, reporter, time, number (number of teller used in reporting).

Responsibilities and collaborators:

- Creation - implemented by
 - number: anInteger reporter: aBankSimulation.
- Updating - implemented by
 - addCustomer: aCustomer. Adds Customer and calculates release time. Collaborates with Customer.
 - updateTime. Checks whether to release Customer, reports when Customer released. Collaborates with BankSimulator.

Implementation

The implementation is simple and we will limit ourselves to a few methods. The rest is left as an exercise.

Output of customer transfers to the Transcript

A typical simulation will run for many units of time and generate a lot of output to the Transcript if the *Show steps* check box is on. This can be relatively time consuming because individual show: messages consume long time. In cases like these, it is better to accumulate the output in the Transcript object (a TextCollector) using nextPut: aCharacter and nextPutAll: aString messages, and flush all accumulated output at the end. We will illustrate this technique on the example of customer dispatch to tellers by Queue.

When Queue contains customers and there is an available teller, it sends a customer using the following message. As a part of this message it informs reporter, the active BankSimulation object:

customer: aCustomer

"A customer has arrived.

customer := aCustomer.

customerReleaseTime := customer processingTime + time.

reporter displayEvent: 'Time: ', time printString, ' Teller ', number printString, '
new customer with processing time ', customer processingTime printString

steps: true

The message to reporter uses method `displayEvent: aString steps: aBoolean` as general purpose event notification message. The method is defined in `BankSimulation` as follows:

displayEvent: aString steps: aBoolean

"Send aString to Transcript unless this is one of the detailed steps and display of steps is not desired."
(showSteps value or: [aBoolean not])
ifTrue: [*Transcript nextPut: Character cr; nextPutAll: aString*]

Since the message to Transcript is not `show:`, the event notification string is not displayed at this point but only stored for future input in Transcript. At the end of simulation, the `SimulationManager` sends message `flush` to `BankSimulation` which then sends the `flush` message to the Transcript. This causes all accumulated output to be output in Transcript in one chunk, a much more efficient operation than a sequence of `show:` messages.

Main lessons learned:

- A queue is a first-in first-out structure (FIFO). Elements are added at one end (tail) and removed from the other end (head).
- Queue behavior is subsumed by `OrderedCollection` and there is no need for class `Queue`.

Exercises

1. Implement class `Queue` with only the essential stack behavior.
2. Implement bank simulation as described.
3. Reimplement bank simulation using polling instead of event-driven design.
4. Modify bank simulation by adding output to file.
5. Repeat our simulation using Poisson's distribution for all random events in the simulation. This distribution is based on the assumption that there is no limit on inter-arrival time.
6. Implement cash register simulation where customers line up in front of individual registers.
7. Explain why queues are also called First-In-First-Out (or FIFO) objects while stacks are called First-In-Last-Out objects.

11.5 Text filter - a new implementation

In this section, we will show how the text filter from the previous chapter can be implemented in a more sophisticated way using a queue.

Problem: The behavior of our previous solution was unsatisfactory and we will thus try to find a better specification and a better solution. Let's try this: Class `TextFilter` takes an original `String` object and replaces occurrences of match substrings with corresponding replacement substrings, compressing the original as much as possible.

Scenario 1: Filtering 'abcdeab' using match/replacement pairs pair1 = 'ab'->'xx', pair2 = 'eab'->'yy'

Assume original string = 'abcdeab' and match/replacement pairs pair1 = 'ab'->'xx', pair2 = 'eab'->'yy'. This is the scenario that failed in our previous implementation. We will now show how the scenario is executed assuming that we keep track of the current position in the original and in each match string via a position counter, and that the resulting string is held in `result`.

1. Initialize `result` to empty string; initialize position counters to 1.
2. Compare position 1 in pair1 (\$a) with first character of string ; match found, increment pair1 pointer.
3. Compare position 1 in pair2 (\$e) with first character of string key; no match.

4. No complete match, copy character from string to result, increment position in string and result.
5. Compare position 2 in pair1 (\$b) with first character of string - match.
6. Compare position 1 in pair2 (\$a) with first character of string key; no match.
7. We have a complete match. Select the longest replacement possible, changing result to 'xx'.
8. Continue in this way until the last character in string. At this point, there are two matches and two possible replacements. The longer one is selected giving result = 'xxcdyy' as expected.

This scenario works as expected. But wait - the behavior in the following scenario does not quite meet our expectations.

Scenario 2: Filtering string = 'abcd' with match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'

Assume string = 'abcd' and match/replacement pairs pair1 = 'bc'->'xx', pair2 = 'abcd'->'yy'. We would assume to obtain the most compressed result = 'yy' but our position counter-based implementation will produce 'axxcd' (left as an exercise).

The reason for the unsatisfactory behavior is that 'bc' is finished matching before 'abcd', and 'abcd' thus never gets a chance to complete matching and be replaced. We thus need a new approach and the best place to start is to analyze why our current approach fails. Our approach does not work because as soon as a substring is matched, it is immediately replaced. However, we cannot make a replacement unless we know that there is no matching in progress that started earlier than the matching that just succeeded, and that is not yet finished.

To deal with this situation, we will use a *queue* of all matching processes in progress such that the matching or matchings that started first are at the head of the queue. The idea is as follows: Match the original character by character. For each character, put all (match string -> replacement string) associations that succeed in matching their first character into a collection and add the collection to the end of the queue. (The queue thus consists of collection objects.) When an association fails to match, remove it from the queue. When a complete match occurs, check whether the matching association is in the collection at the head of the queue. If it is, there is no earlier matching in progress and we can thus safely make the replacement in result. At this point, we will empty the whole queue and start a new matching sequence. If the matching association is not in the collection at the head of the queue, mark it as ready for replacement and proceed; don't do any replacement yet - wait until the association reaches the beginning of the queue. When it does, use it to make the appropriate replacement.

Before we formulate the algorithm more carefully, we will take a look at the objects needed to support it. In addition to streams and position counters to keep track of the original string and the result, we also need

- A queue whose elements are collections of partially matched associations. We will call it `MatchesInWaiting` and implement it as a class variable.
- A collection of all associations that are fully matched and ready for replacement. This will be a class variable called `ReadyForReplacement`.

With this background, we can fully describe the matching procedure as follows:

1. Create a `ReadStream` over the original string and a `WriteStream` over the string being constructed. Create an empty `MatchesInWaiting` queue. Create a `MatchDictionary` and initialize it to *match string -> two-element array* associations; each association has a match string for key. The first element of its value array is the replacement string, the second element of the array is the position counter initialized to 0.
2. Repeat for each position of the input stream beginning from the start:
 - a. Add a new collection to the end of the `MatchesInWaiting` queue.
 - b. For each element of the dictionary do:
 - Increment position counter of this dictionary item.

Compare original character and match character.

If no match, reset current position counter of dictionary item to 0. If this item is in MatchesInWaiting, remove it.

If match, check if this is match on first character of match string.

If so,

add this association to the collection at the end of the MatchesInWaiting queue.

Check if this is the last character to match.

If so (match succeeded), check if ReadyReplacement is empty.

If so, store this association in ReadyReplacement.

If this is not the last character (match incomplete), increment position counter of this association.

- c. If ReadyReplacement contains an association and if this association is at the head of the queue, use the association to make a replacement in OutputStream.

Empty MatchesInWaiting queue, reset ReadyReplacement and ReplacementPosition to nil.

We leave it to you to check whether this algorithm works, correct it if it does not, and implement it.

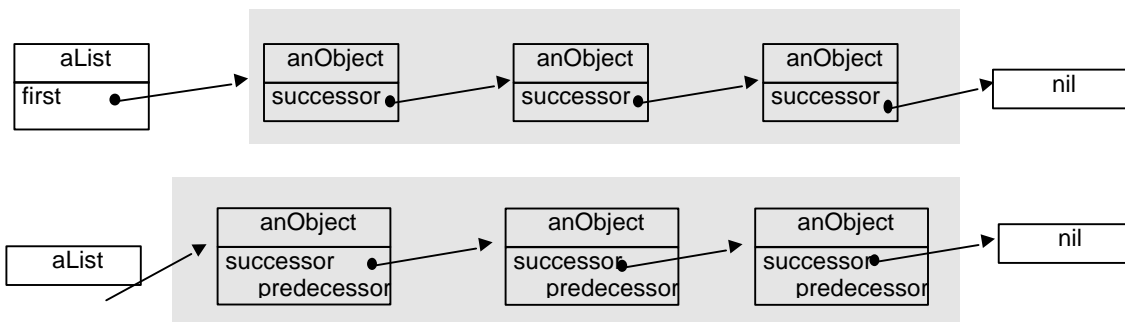
Exercises

1. Complete the design of the text filter and implement and test it.
2. Our algorithm will work if all the replacement strings are equally long. But what if they are not?
3. Extend the filter to allow replacement calculated by blocks.
4. Our various formulations of the string replacement problem were not incorrect but merely different. Are there any other possible formulations of the string replacement problem? If so, outline the appropriate solutions.
5. Since there are several possible formulations of text filtering that each require a different solution, the problem seems amenable to an abstract class with concrete subclasses implementing different specifications. Design such a hierarchy and comment on the advantages of this approach - if any.

11.5 Linked Lists

None of the sequenceable collections covered so far are specifically designed for insertion of new elements at arbitrary locations. Arrays allow replacement but not insertion, and ordered collections allow insertion but the internal operation is complex and inefficient.

A linked list is a collection of objects linked to one another like a string of pearls. As a minimum, a linked list knows about its first element, and each element knows about its successor. A *doubly* linked list is an extension of a singly linked list whose elements also know about their predecessor (Figure 11.10). Both provide a way insert a new element at any location. Another advantage of linked lists is that they occupy only the amount of space required to hold their elements whereas ordered collections may occupy more space because they are normally only partially filled. However, elements of linked lists (their *nodes*) must be packaged in more complex objects.



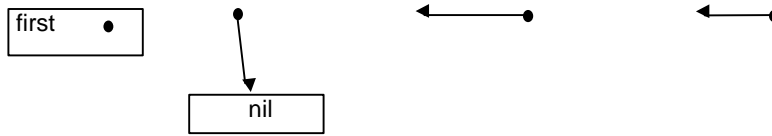


Figure 11.10. Linked list (top), doubly linked list (bottom). List elements are shaded.

As our illustration shows, the implementation of linked lists requires two kinds of objects - list elements and a linked list object itself. A minimal linked list must know at least about its first node. In the Smalltalk library, linked lists are implemented by class `LinkedList` with the following comment:

The class `LinkedList` implements ordered collections using a chain of elements. Each element of a `LinkedList` must be an instance of class `Link` or of one of its subclasses.

A new instance of `LinkedList` can be initialized using

```
LinkedList with: Link new
Instance Variables:
    firstLink    <Link>
    lastLink     <Link>
```

Class `LinkedList` is a subclass of `SequenceableCollection` and inherits its protocol including creation, enumeration, and access by index (normally unused), and redefines many of them.

Class `Link` whose instances are nodes of `LinkedList` is very primitive and implements only linking with no provision to hold a value. Its comment is

Class `Link` represents a simple record of a pointer to another `Link`.

```
Instance Variables:
    nextLink     <Link> a pointer referencing the next Link in the chain
```

`Link` protocol includes `nextLink` (returns the next node in a linked list), `nextLink: aLink` (sets `nextLink` to specified `Link` object), and the creation message `nextLink:` that initializes the successor of the receiver. Being so primitive, `Link` is only used as a superclass of classes that attach a value to the link object, thus allowing the creation of `LinkedList` objects with non-trivial elements as in Figure 11.10.

Linked lists are simple but useful objects and we will illustrate their use on the following example:

Example. Reimplement sorted collections using linked lists

Problem: Use a linked list to implement class `SortedLinkedList` with the functionality of `SortedCollection`. Restrict the definition to creation and adding.

Solution: Being an extension of `LinkedList`, `SortedLinkedList` will be a subclass of `LinkedList` with a new instance variable called `sortBlock`. The nature of the sort block will be the same as that of `SortedCollection` and there will also be a default sort block stored in a class variable `DefaultSortBlock`. Its value will be `[x :y | x <= y]` so that a `SortedLinkedList` with the default block will have its elements sorted in the order of ascending magnitude of its elements. The new message will create a `SortedLinkedList` with the default sort block, and an instance with a different sort block can be created by the class method `sortBlock: aSortBlock`. The definition of the class is

```
LinkedList subclass: #SortedLinkedList
  instanceVariableNames: 'sortBlock '
  classVariableNames: 'DefaultSortBlock '
  poolDictionaries: ''
  category: 'Book'
```

The value of the default sort block will be assigned by the same *class* initialization method as in SortedCollection:

```
initialize
"Create a default sort block"
DefaultSortBlock := [:x :y | x <= y]
```

Don't forget to execute the initialization method with

SortedLinkedList initialize

before using SortedLinkedList for the first time. We suggest using *inspect* to see the effect.

Next, the creation method. The method creates a new instance and then initializes the sort block by

```
new
  ^super new initialize
```

where the *instance* method initialize is again the same as in SortedCollection:

```
initialize
"Set the initial value of the receiver's sorting algorithm to a default."
sortBlock := DefaultSortBlock
```

When we test the code developed so far with

SortedLinkedList new

we get the expected result – try it with *inspect*.

Before we can implement adding, we must be able to create node objects. Since the nodes in our list have a value, they cannot be simple Link objects. We will thus create a new class called LinkNode with instance variable value, extending Link with creation method nodeValue: anObject, and accessing methods value and value: anObject. We will leave these methods as an exercise and limit ourselves to comparison by <= which is required by the sort block:

```
<= aNode
"Compare the value of the receiver and the value of the argument."
^self value <= aNode value
```

We suggest that you test that LinkNode methods work.

Now that we have a class to implement nodes, we can return to SortedLinkedList. We will limit ourselves to method add: anObject for adding a new object to a linked list and leave other functionality as an exercise. The add: message will be based on careful consideration of all possible situations expressed in the following algorithm:

1. If the list is empty, add the new node using the inherited add: message and exit.
2. If the list is not empty, search it from the beginning looking for a node that returns false when compared with the new node using the sort block.
 - a. If such a node is found, insert the new node before this element and exit.

- b. If such a node is not found, add the new node at the end of the list

We will now examine Step 2 in more detail. Each evaluation of the sort block has two possible outcomes: If the block returns true, we must continue the search and proceed to the next node. If the block returns false, the current node is either at the beginning of the linked list or inside it (Figure 11.11). If the current node is the first node in the list, the new node must be inserted in front of it using the inherited `addFirst:` message. If the current node is not the first node, the new node must become the `nextLink` of the previous node, and its own `nextLink` must point to the 'current' node. This means that the method must keep track of the previous node.

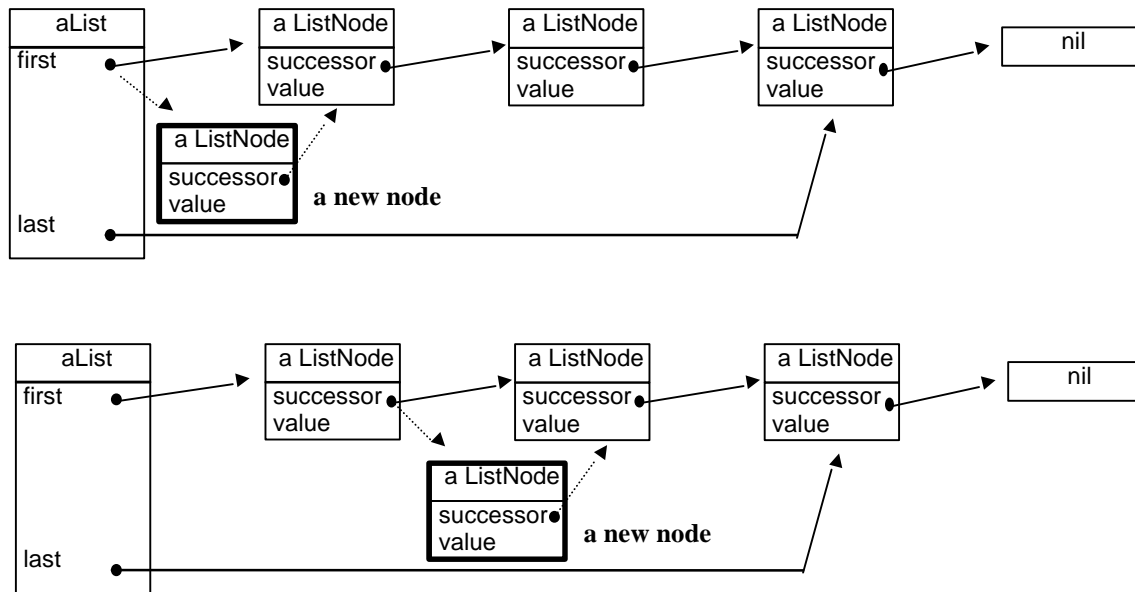


Figure 11.11. When the list is not empty, the new node will be inserted either at the start of the list (top), or into the list (bottom), or behind the last element (not shown).

With this background, the definition of `add:` can be written as follows:

add: anObject

"Add anObject to the list at position determined by sortBlock. Return anObject."

```
| current newNode previous |
"Create a node link with value anObject."
newNode := LinkNode nodeValue: anObject.
"If the list is empty, simply put the node at the start of the list."
self isEmpty ifTrue: [super add: newNode. ^anObject].
"List is not empty, traverse it from the start."
current := self first.
[current isNil] "End of list?"
whileFalse: [
    (sortBlock value: current value: newNode) "Is new node behind current node?"
    ifTrue: "Sort block succeeded, go to next list node."
        [previous := current.
         current := current nextLink]
    ifFalse: "Sort block failed, distinguish two possibilities."
        [current == firstLink
         ifTrue: "Insert new node at the start of the list."
             [newNode nextLink: firstLink.
              self addFirst: newNode]
```

```

                                ifFalse: "Insert between previous and current."
                                    [previous nextLink: newNode.
                                     newNode nextLink: current].
                                ^anObject]].
    "Node does not fit inside the linked list, add it at the end."
    self addLast: newNode.
    previous nextLink: newNode.
    ^anObject
    
```

We are now ready for testing. We cannot test every possible situation but we must try to test every *logical* possibility. We will test add: in the following situations:

1. List is empty.
2. List is not empty and the new node becomes the first node in the list.
3. List is not empty and the new node is inserted between two existing nodes.
4. List is not empty and the new node is added at the end of the list

An example of a test program that checks all these alternatives is

```

| sll |
Transcript clear.
sll := SortedLinkedList new.
Transcript show: sll printString; cr.
sll add: 3.
Transcript show: sll printString; cr.
sll add: 33.
Transcript show: sll printString; cr.
sll add: 13.
Transcript show: sll printString; cr.
sll add: 0.
Transcript show: sll printString
    
```

Before we can execute this program, we must make sure that printString creates meaningful SortedLinkedList descriptions. Since SortedLinkedList is a subclass of Collection, we reuse the printing protocol in Collection which depends on do:. We must thus define do: as follows:

do: aBlock

```

"Evaluate aBlock with each of the receiver's elements as the argument."
current := firstLink.                                "Initialize for traversal."
[current isNil]                                        "Iterate up to and including the last node."
whileFalse:
    [aBlock value: current.
     current := current nextLink].                    "Get the next node, if any."
    
```

With this method, printing works but does not produce any meaningful information about nodes. We define node printing as

printOn: aStream

```

"Append to the argument, aStream, the elements of the Array enclosed by parentheses."
aStream nextPutAll: self class name; nextPutAll: ' '; nextPutAll: self value printString
    
```

and with this, our test program produces the following correct output which confirms that our definitions work as expected:

```

SortedLinkedList ()
SortedLinkedList (LinkNode 3)
SortedLinkedList (LinkNode 3 LinkNode 33)
    
```

```
SortedList (LinkNode 3 LinkNode 13 LinkNode 33)  
SortedList (LinkNode 0 LinkNode 3 LinkNode 13 LinkNode 33)
```

Finally, we must test that SortedLinkedList works even with non-default sort blocks and we thus define a new SortedLinkedList creation method called sortBlock: and modify our test program to create a new list with the same elements sorted in *descending* order:

```
| sll |  
Transcript clear.  
sll := SortedLinkedList sortBlock: [:x :y | x > y].  
Transcript show: sll printString; cr.  
sll add: 3.  
Transcript show: sll printString; cr.  
sll add: 33.  
Transcript show: sll printString; cr.  
sll add: 13.  
Transcript show: sll printString; cr.  
sll add: 0.  
Transcript show: sll printString
```

The test produces

```
SortedList ()  
SortedList (LinkNode LinkNode 3)  
SortedList (LinkNode LinkNode 33 LinkNode LinkNode 3)  
SortedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3)  
SortedList (LinkNode LinkNode 33 LinkNode LinkNode 13 LinkNode LinkNode 3 LinkNode LinkNode 0)
```

in the Transcript and again confirms the correctness of our implementation.

Main lessons learned:

- A linked list is a linear collection of nodes with a pointer to its first element and links between consecutive nodes.
- In addition to forward links, linked nodes can also have backward links. Such lists are called doubly linked lists.
- A linked list allows insertion of elements at any point in the list.
- Smalltalk library contains a pair of general classes called LinkedList and Link implementing the basic linked list behavior. For concrete use, these classes must be subclassed.

Exercises

1. Complete the implementation of the example from this section and add node removal.
2. Find how an element can be inserted into an OrderedCollection.
3. Test which implementation of sorted collection is faster using the Speed Profiler included in the Advanced Tools of VisualWorks. Justify the result.
4. Compare the memory requirements of the two implementations of sorted collection using the Allocation Profiler included in the Advanced Tools of VisualWorks.
5. Add node removal to sorted linked list.
6. Add method asArray returning an Array containing the values of all elements of the list in the order in which they are stored.
7. Implement a doubly linked list.

11.6 Trees

A tree is a two-dimensional collection of objects called nodes. Three examples are a class hierarchy tree, a family tree, and a tree of student records as in Figure 11.12. The nodes of a tree can be classified as follows:

- The node at the top of the tree is called the *root*. This is the node through which the tree can be accessed. The node with ID 27 in our example is the root of the tree. A tree has exactly one root.
- A node that is not a root and has at least one child is an *internal* node. The records with IDs 13 and 32 in our example are internal nodes. Every internal node has exactly one parent node but it may have any number of children, unless the definition of the tree specifies otherwise.
- A node that does not have any children is called a *leaf*. The node with ID 11 in our example is a leaf node. Like an internal node, a leaf has exactly one parent node.

An interesting and useful property of trees is that any node in the tree can be treated as the root of a new tree consisting of all the underlying nodes. Such a tree is called a *subtree* of the original tree. As an example, the node with ID 32 in Figure 11.12 can be used as the root of a tree with children 29 and 45. Leaves, such as 29, can be thought of as degenerate trees that consist of a root and nothing else.

Trees are very important in computer applications and a variety of trees have been devised to provide the best possible performance for different uses. The differences between different kinds of trees are in the number of children that a node may have⁴, and the way in which the tree is managed (how nodes are added and deleted). The subject of trees is not trivial and we will restrict our presentation to the example of a simple binary tree. In a binary tree, each node may have at most two children as in Figure 11.12.

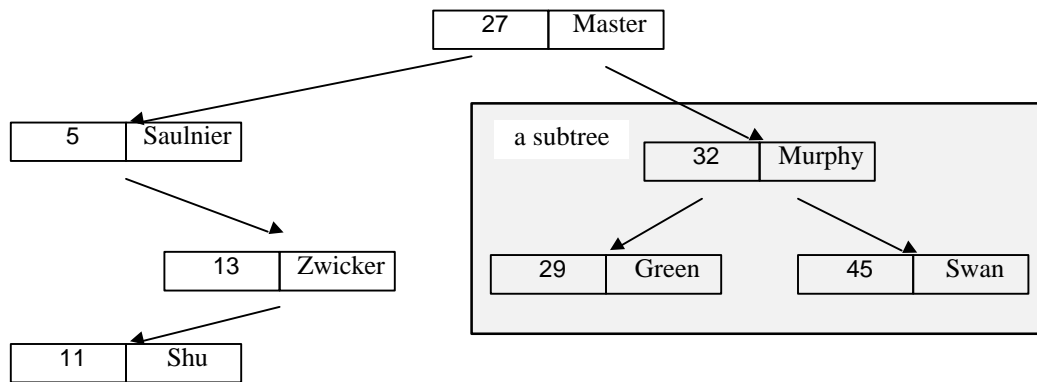


Figure 11.12. Binary tree of student records based on comparison of student IDs.

Problem: Implement a binary tree whose nodes are arranged on the basis of comparison of node values. The tree must be capable of adding a new object, deleting an existing node, and finding whether an object with a given value is in the tree. Provide enumeration and printing as well.

Solution: Our implementation will use two kinds of objects - nodes (class `Node`), and the tree itself (class `BinaryTree`). The `BinaryTree` object must know about its root and how to perform comparison (comparison block), the `Node` object must know its value and its children; knowing the parent may also be useful. We will start with `Node` because we cannot create `BinaryTree` without nodes. The *comment* of `Node` is as follows:

Class `Node` implements nodes for class `BinaryTree`. Each node consists of

⁴ Each node with the exception of the root always has exactly one parent.

value	<Object>	- value of node, must understand the tree's comparison block
leftChild	<Node nil>	- root of left sub-tree of this node, a Node or nil
rightChild	<Node nil>	- root of right sub-tree of this node, a Node or nil
parent	<Node nil>	- parent of this node, a Node or nil

The superclass of Node will be Object and we will start its implementation with a class method to create a Node with its value. The method is styled in the usual way:

```
newWithObject: anObject  
  ^self new value: anObject
```

and instance method value: sets value to anObject. We leave the remaining instance variables uninitialized which is as it should be because they will be assigned as new objects are added to the tree. The methods to add children and the parent are equally simple but we must remember that when a node is added as a child, the receiver node must become the child's parent as in

```
rightChild: aNode  
  rightChild := aNode.  
  aNode parent: self
```

After defining all accessing methods, we can now test our design by executing the following code with *inspect*:

```
|parentNode node nodeOnLeft nodeOnRight|  
parentNode := Node newWithObject: 'parent node'.  
node := Node newWithObject: 'this node'.  
nodeOnLeft := Node newWithObject: 'left sub-node'.  
nodeOnRight := Node newWithObject: 'right sub-node'.  
node parent: parentNode.  
node leftChild: nodeOnLeft.  
node rightChild: nodeOnRight.  
node
```

We find that everything is OK.

Now that we have a working Node, we can start working on class BinaryTree. Each instance of BinaryTree needs to know the tree's root node and its comparison block which is used to decide whether a new node should be inserted in the left subtree or the right subtree. We will call these two instance variables root and isLeftBlock and comment the class as follows:

Class BinaryTree implements a binary tree. Nodes are added as leaves on the basis of a comparison block applied to the value of the new object and values of existing nodes.

Instance variables

root	<Node nil>	- root node of the tree, a Node or nil.
isLeftBlock	<BlockClosure>	- a two-argument block used to determine whether a new node should be added to the left of an existing node. The first argument of the block is the new node, the second argument is an existing node to which the new node is being compared. When the block returns true, the new node is attached as the left subnode; on false, the new node is attached on the right.

The superclass of BinaryTree is Object. The *creation* method will specify the root of the tree and the comparison block. It will be used as in

```
BinaryTree withRoot: 13 withBlock: [:new :old| new < old]
```

The creation method is as follows:

withRoot: anObject withBlock: comparisonBlock

```
^self new root: (Node newWithObject: anObject) block: comparisonBlock
```

where instance method `root:block:` initializes `root` and `isLeftBlock`. Implement these two methods and test everything by executing the above expression with *inspect*.

Now that we can create a tree with a root and a comparison block, we need a method to add an object to an existing tree. To add a new object, we start from the root and proceed downwards using the following principle: If the value of the new object satisfies the comparison block with the value of the current node, proceed to the left child, if any; otherwise proceed right. Upon reaching a node whose desired child is nil, further traversal is impossible; add the new object as a new leaf on the left or on the right on the basis of comparison of comparison.

Since adding works the same way for the root and for sub-nodes, the `add:` method can take any node as its argument and treat it as the root of a tree and re-send itself with this node as the root. A more accurate description of the algorithm is as follows:

1. If the tree is empty, make the new object the root of the tree and exit.
2. If the tree is not empty, proceed as follows:
 - a. If the root and the new object satisfy the sort block, send `add:` to its left subtree.
 - b. If evaluation of the sort block returns `false`, send `add:` to the right subtree.

You can check that the tree in Figure 11.12 is the result of this algorithm with nodes added in the following order of IDs: 27, 13, 8, 32, 29, 11, 45. Our description is easily translated into the following code:

add: newNode toNode: existingNode

```
"Attach newNode as leaf of the appropriate subtree of existingNode."  
existingNode isNil ifTrue: [root := newNode. ^newNode].  
(isLeftBlock value: newNode value value: existingNode value)  
ifTrue: [ "Add newNode to left sub-tree."  
existingNode leftChild isNil  
ifTrue: "Attach aNode as leftChild."  
[existingNode addLeftChild: newNode]  
ifFalse: "Apply this method recursively."  
[self add: newNode toNode: existingNode leftChild]]  
ifFalse: "Add newNode to right sub-tree."  
[existingNode rightChild isNil  
ifTrue: "Attach aNode as rightChild."  
[existingNode addRightChild: newNode]  
ifFalse: "Apply this method recursively."  
[self add: newNode toNode: existingNode rightChild]].  
^newNode
```

Note that we used `value: value:` to evaluate the comparison block because the block has two arguments.

This method can be used to add a node once we have a node to start from. First, however, we must get into the tree and get the starting node – the root. To do this, we will use the following method:

add: anObject

```
"Add node with anObject as a new leaf of the tree."  
self add: (Node newWithObject: anObject) toNode: root
```

We can now test our solution with the following program which should create the tree in Figure 11.13:

```
|tree |  
tree := BinaryTree withRoot: 15 withBlock: [:new :old| new value < old value].  
tree add: 8;  
    add: 21;  
    add: 14;  
    add: 36;  
    add: 17;  
    add: 49;  
    add: 32;  
yourself
```

Execute the program with *inspect* and check that the resulting BinaryTree is correct. Note that once you open the tree inspector, you can use it to traverse the whole tree by selecting first the root and then inspecting its children, proceeding recursively in this way as far as you wish.

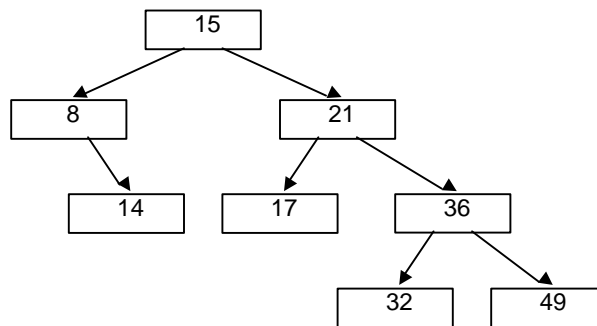


Figure 11.13. Binary tree created with the program in the text.

Unfortunately, our `add:` method ignores the possibility that the root is `nil` which will happen if we deleted all nodes. This can be corrected as follows:

```
add: anObject  
"Attach node with anObject as a new leaf of the tree."  
| newNode |  
newNode := Node newWithObject: anObject.  
root isNil  
    ifTrue: [root := newNode]  
    ifFalse: [self add: newNode toNode: root]
```

The next protocol is finding. The `find: anObject` method will return the node whose value is `anObject`, or `nil` if such a node is not in the tree. We leave it as an exercise and suggest that you implement it via two methods as we did for adding.

To *delete* a node with a specified value, we must find the node, remove it from the tree, and rearrange the tree so that new node ordering still satisfies the sort block. The new tree may not look like the original tree but the relation between a node and its children must satisfy the comparison block.

The easiest (but not the most efficient) way to implement this specification is to remove the node and then add all its children and the remaining nodes to its parent using the *add* operation. By definition, this will preserve proper ordering. A detailed description of the deletion algorithm is as follows:

1. Find node `node` whose value is `anObject`. If there is no such node, execute an exception block and exit.
2. If node is root, then
 - a. If its left child is `nil` then
 - i) Replace root with left child.
 - ii) Replace new root's parent with `nil` to cut the link to the removed root.
 - b. Else (the left child is not `nil`)

- i) Add root's right child to the root's left child.
 - ii) Make left child the new root.
 - iii) Replace new root's parent with nil to cut the link to the removed root.
- 3. If node is not root then
 - a. Remove connection from node's parent to node.
 - b. Add the left child of node to the parent – this adds all left subnodes because it leaves the child's subtrees unchanged.
 - c. Add the right child of node to the parent– this adds all right subnodes because it leaves the child's subtrees unchanged.

This algorithm can be used to define delete as follows:

remove: anObject ifAbsent: aBlock

"Delete anObject from the tree and return anObject; evaluate aBlock if not present."

```
| parent theNode |
theNode := self find: anObject.
theNode isNil ifTrue: [^aBlock value].
parent := theNode parent.
parent isNil
    ifTrue: ["theNode must be the root."
        root leftChild isNil
            ifTrue: [root := root rightChild]
            ifFalse: [self add: root rightChild toNode: root leftChild.
                root := root leftChild].
        root parent: nil]
    ifFalse: [parent leftChild = theNode
        ifTrue: [self removeLeftChild]
        ifFalse: [self removeRightChild].
        self add: theNode leftChild toNode: parent.
        self add: theNode rightChild toNode: parent].
^anObject
```

where removeLeftChild and removeRightChild simply replace the child with nil. Finally, *enumeration* can be implemented by a combination of two methods:

do: aBlock

"Evaluate aBlock for each node of the tree rooted at the root."

```
self do: aBlock startingFrom: root
```

and

do: aBlock startingFrom: aNode

"Evaluate aBlock for each node of the tree rooted at aNode."

```
aNode isNil
    ifFalse: [aBlock value: aNode.
        self do: aBlock startingFrom: aNode leftChild.
        self do: aBlock startingFrom: aNode rightChild]
```

This very concise method is a typical example of the advantage of using recursion to implement a non-trivial operation on a complicated collection. The method can be tested, for example, by

```
|tree |
tree := BinaryTree withRoot: 15 withBlock: [:new :old| new value < old value].
tree add: 8;
add: 21;
add: 14;
add: 36;
```



```
add: 17;  
add: 49;  
add: 32;  
do: [:node | Transcript show: node value printString; cr]
```

which produces

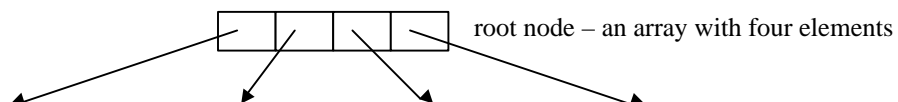
```
8  
14  
21  
17  
36  
32  
49
```

Main lessons learned:

- A tree is a branching structure of nodes and their children emanating from one node called the root.
- Nodes at the bottom of the tree (nodes with no children) are called leaves.
- Every node in a tree except the root has exactly one parent node. The root does not have a parent.
- Specialized trees may restrict the number of children of a node or the structure of the tree.

Exercises

1. List two different number sequences that will produce the tree in Figure 11.13.
2. Complete and test `BinaryTree`.
3. How could we improve our design of `BinaryTree`? One weakness of our approach is that when we traverse the tree, we check each node to determine whether it is empty or not because an empty node requires different treatment. This overhead is unnecessary if we use polymorphism and implement `BinaryTree` as an abstract class with two subclasses – one for an empty binary tree and one for a non-empty tree. Implement this idea (borrowed from [Pugh, Lalonde]) and compare the efficiency of the two approaches.
4. Add the following methods to the implementation of binary tree:
 - a. A method to merge two trees together.
 - b. A method to check whether a tree contains another tree as its subtree.
 - c. A method to copy nodes from one tree to another tree.
5. Assume that in addition to binary trees, we also want to implement trees with up to three children, and trees with up to four children. Both trees use some further unspecified construction rules. Is it desirable to define an abstract superclass? If so describe its properties in detail. What other classes must be defined?
6. Explain in what order the `do:startingFrom:` method traverses the nodes of the tree. Can you describe other traversals? Implement them.
7. Add a method to find the depth of the tree - the maximum number of nodes encountered during the traversal from the root to a leaf.
8. Consider $n = 255$ objects sorted according to their value. What is the maximum number of steps required to find a given object if the objects are stored in a sorted collection? What is the answer if the objects are stored in a balanced tree whose branches have, as much as possible, equal depth? What is the answer for a general value of n ?
9. A two-dimensional array can be implemented with a tree as in Figure 11.14. Describe how this principle could be used to implement an n -dimensional array.





leaf – an array with three elements

Figure 11.14. Representing a two-dimensional 4x3 matrix by a two-level tree. Leaves represent rows.

11.7 Use of trees in compilation

One of the most important uses of trees is in compilation. When you compile a method with *accept*, the following sequence of events occurs:

1. A message is sent to the **Compiler** object with the source code as one of its arguments.
2. The compiler sends a message to the **Parser** object to find the structure of the method.
3. The parser sends a message to the *lexical analyzer* which converts the source code string into *tokens* - symbols representing higher level language entities such as literal numbers or strings, and unary or binary selectors.
4. The parser uses the tokens and the definition of Smalltalk syntax to construct the *parse tree* representing the source code.
5. The *code generator*, reads the parse tree, and produces the translation - an instance of **CompiledMethod**.

The process is, of course, interrupted if the code contains illegal source code. We will now illustrate the operation on an example. Assume that you entered the following method in class **Test** using the Browser and clicked *accept*.

test

```
"A test to illustrate compilation."
3 + 3.
5 factorial
```

If you trace the execution of the ensuing compilation by inserting a breakpoint in front of the statement in Step 1 (below), you will find that the steps listed above take the following form:

1. The compiler is invoked with the message

```
Compiler new compile: 'test 3 + 3. 5 factorial'
  in: Test
  notifying: nil
  ifFail: []
```

2. The compiler invokes the parser with the message

```
Parser parse: aStream
  class: class
  noPattern: noPattern
  context: context
  notifying: handler
  builder: ProgramNodeBuilder new
  saveComments: mapFlag
  ifFail: [^failBlock value]
```

where **aStream** contains the source code of the method, and the remaining arguments specify who will build the parse tree and provide additional information.

3. The parser asks the scanner to convert the source code into tokens. In our case, the scanner produces the following sequence of token-type/token-value pairs:

#word		'test'
#number	3	
#binary		+
#number	3	
#period		.
#number	5	
#word		'factorial'

In constructing this sequence, the scanner skipped all characters that don't have any execution effect such as spaces, tabs, and carriage returns. (In actual operation, the scanner transfers control to the parser after each token-type/token-value pair but the principle is the same as if all tokens were generated first which is what we will assume for ease of presentation.)

4. The parser converts the tokens into a parse tree. In terms of Smalltalk objects, the parse tree is one big nested object, the MethodNode object shown in Figures 11.15.

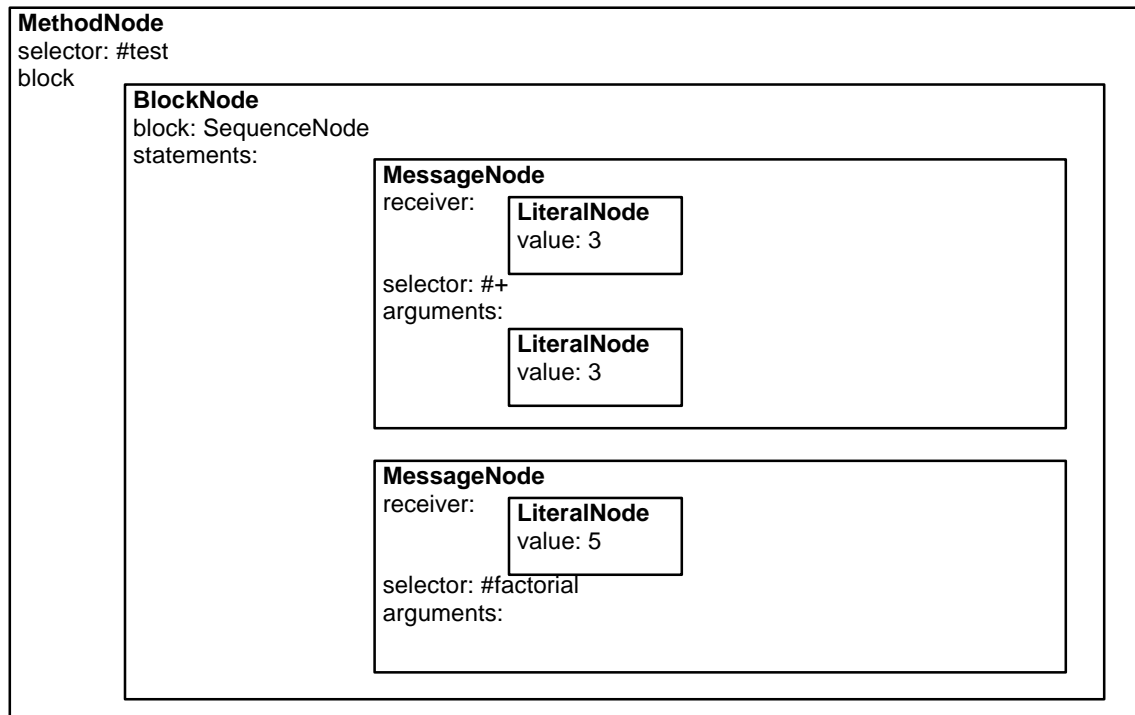


Figure 11.15. Nested MethodNode object produced for the example method in the text. Only the essential attributes are shown.

The methodNode object can be represented as the *parse tree* in Figure 11.16. Note that each node may be a different kind of object with different instance variables, that there is no prescribed number of children, and that the structure of the tree is determined by the structure of the source code and the syntax rules of Smalltalk rather than comparison of node values as in our previous example.

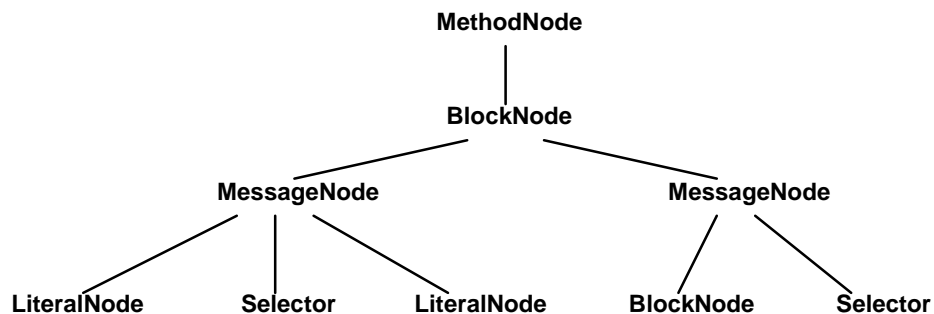


Figure 11.16. Parse tree corresponding to Figure 11.15.

5. Finally, the code generator uses the parse tree to calculate the following CompiledMethod:

normal CompiledMethod numArgs=0 numTemps=0 frameSize=12

literals: (#factorial)

```
1 <D8 03> push 3
3 <D8 03> push 3
5 <DF 00> no-check send +
7 <66> pop
8 <D8 05> push 5
10 <CC 00> no-check send factorial
12 <45> pop; push self
13 <65> return
```

In conclusion, let's note that both Parser and Scanner are in the library and you can subclass them to create recognizer for objects defined by their syntax or to parse programs written in another programming language.

Main lessons learned:

- Compilation consists of lexical analysis (scanning), parsing, and code generation.
- Lexical analysis extracts program tokens from the source code, parsing recognizes the grammatical structure of the program, code generation produces a CompiledMethod with executable bytecodes.
- Parsing explicitly or implicitly creates a parse tree.

Exercises

1. Create a simple method, trace its execution, and describe the process.

11.8 Graphs

A graph is a mathematical abstraction of concepts such as the road map (Figure 11.17), the layout of house plumbing, the telephone network, the structure of course prerequisites, or the world-wide web. A graph can be completely described by listing its *nodes* (usually called *vertices* - the plural of *vertex*) and the *edges* (*arcs*) connecting them. When the edges are labeled with numbers representing, for example, distances or driving times between cities on a road map, the graph is called a *weighted graph* and edge weights must be included in the description of the graph. As an example, our road map can be described as follows:

$V = \{\text{Chicago, New York, Toronto, Montreal, Vancouver, San Francisco, Denver, Dallas}\}$ "Set of vertices."
 $E = \{(\text{VT}, 60), (\text{TM}, 6), (\text{VS}, 21), (\text{VD}, 33), (\text{TD}, 31), (\text{TC}, 11), \text{etc.}\}$ "Set of weighted edges."

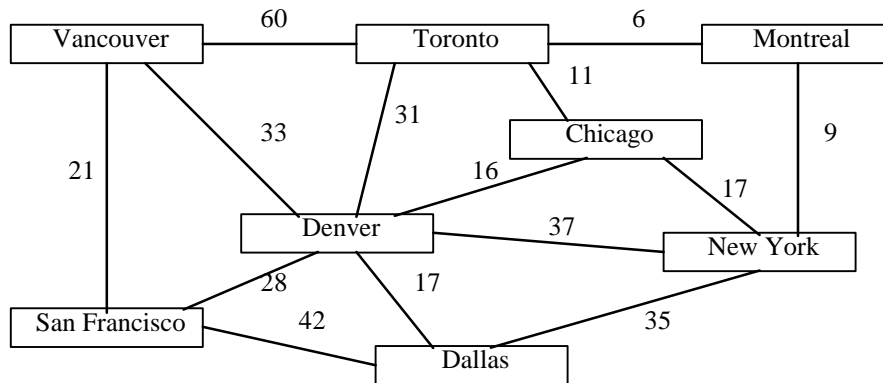


Figure 11.17. Road map with driving times along selected highways.

The graph in Figure 11.14 is an *undirected graph* because its edges don't restrict the direction in which they must be traversed. In other words, the roads between the cities are not one-way roads. A map showing airline connections (Figure 11.18), on the other hand, may have restrictions because the airline may not provide flights in both directions between all cities. Such a graph is called a *directed graph*.

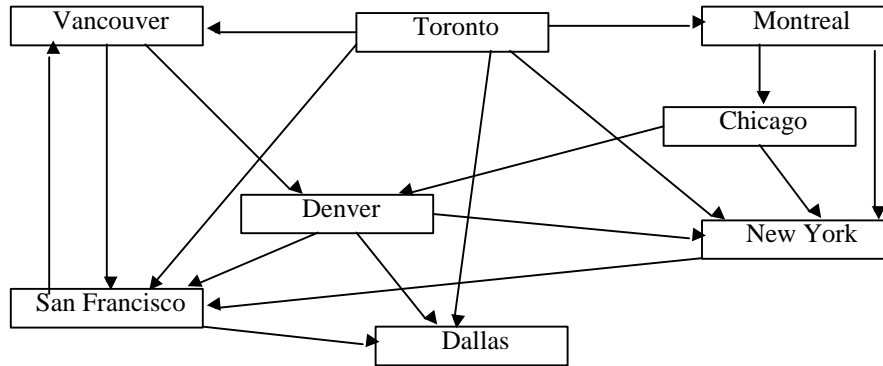


Figure 11.18. Directed graph of fictitious airline connections.

Many books have been written about graphs, their representation, the algorithms for performing various calculations on them, and their relative efficiency. In this section, we will only address the problem of finding the shortest path between two vertices in an undirected graph with weighted edges (the shortest driving time between two cities), finding the shortest path between two vertices in a directed graph with unweighted edges (the minimum number of flight changes), and testing connectivity (existence of a path from one city to another).

Class Graph

All graphs can be represented as directed weighted graphs: Undirected graphs are directed graphs in which all pairs of vertices have either edges in both directions or none, and unweighted graphs can be represented as graphs where all edges have some irrelevant weight, possibly nil. We can thus implement all graphs using just one class whose attributes consist of vertices and weighted edges. This representation is, of course, redundant but we will use it for our examples.

One way to think about a graph is as a set of vertices, each of which has a distinguished value (such as a city name), knows its neighbors (vertices that can be reached from it), and the cost of getting to each neighbor. This is easily implemented by a Dictionary whose keys are vertices and whose values are objects consisting of a connection to a neighbor (edge) and the weight of the connecting edge. We could implement these edge-weight objects as instances of a new class but they are so simple that we will implement them as two-element arrays. As an example, our unweighted airline map could be represented as a dictionary containing the following associations:

```
'Chicago' -> Set ( #('NewYork' nil) #('Denver' nil))
'Dallas' -> Set ()
'Denver' -> Set ( #('Dallas' nil) #('NewYork' nil) #('SanFrancisco' nil))
etc.
```

It is tempting to make the new class Graph a subclass of Dictionary but this would be wrong because the two classes don't share any functionality. We will thus make Graph a subclass of Object and put the Dictionary describing its structure into an instance variable called structure. The following is the comment of the class:

I implement directed graphs consisting of vertices and weighted edges.

Instance variables

structure <Dictionary> association key is node, value is a set of two-element arrays consisting of vertex ID and the corresponding edge weight.

As usual, we will start implementation with the creation protocol. The creation method simply creates a new Graph and assigns a dictionary to structure as follows:

newWith: aDictionary

"Create a new graph whose structure is defined by aDictionary."

^self new dictionary: aDictionary

After creating a printOn: method and testing it (left as an exercise), we now define additional accessing methods. As an example, the following accessing method to simplifies access to neighbors:

edgesFor: anObject

"Return the set of vertex-weight objects of vertex anObject."

^structure at: anObject

We are now ready to deal with the graph problems listed at the beginning of this section.

Testing connectivity

Problem: Define a method to determine whether there is a path from one vertex to another, for example, a sequence of flights from Montreal to San Francisco.

Solution: The principle of the solution is simple (Figure 11.19): First, we find and mark all vertices that can be reached directly from to source vertex (Montreal). We then find and mark all vertices that can be reached directly from these cities. And we continue doing this until we either reach the destination vertex (San Francisco) or until there are no more unmarked reachable cities. In the first case there is a path from Montreal to San Francisco, in the second case there is no such path.

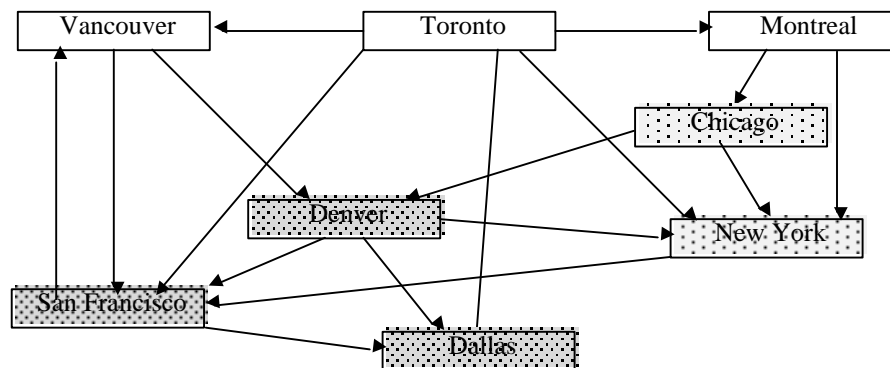


Figure 11.19. Testing connectivity. The source (Montreal) is a white rectangle, adjacent cities are shown with light gray, cities directly reachable from them are shown with darker gray, and so on until we reach San Francisco.

The method implementing this idea requires two arguments – an identification of the source vertex and the destination vertex - and returns true or false. We will call it `connected:to:` and use it as in

flights connected: 'San Francisco' to: 'Toronto'

where flights is presumably the graph describing the flight map.

Our definition will follow the informal description given above but refined down to the level of implementation. The informal description shows that we are always dealing with three kinds of vertices: Those that have already been processed, those that are now being used to find new reachable vertices, and those that have not yet been used at all. This can be implemented by keeping vertices of each kind in a collection. If we call these boxes *done*, *active*, and *remaining*, we can describe the algorithm as follows:

1. Put the starting vertex into *active*, initialize *remaining* to all other vertices, and *done* to empty.
2. Repeat the following until the destination vertex appears in *active*, or until no more vertices are accessible, in other words until *active* becomes empty:
 - a. Enumerate over all vertices in *active*:
 - For each vertex *v*
 - add all its immediate neighbors in *remaining* into *active*
 - move vertex *v* from *active* to *done*.

Using a queue for the collection leads to a *breadth-first algorithm* which first explores all vertices at the same 'level' from the starting situation. An algorithm that follows a path all the way down and then starts another is called a *depth-first algorithm*. We will use a queue for *active* as follows:

connected: object1ID to: object2ID

"Is the object whose ID is object1ID connected to the object whose ID is object2ID? Return true or false."

```
| active done remaining startObject found |
"Initialization."
active := OrderedCollection new: structure size.      "Use OrderedCollection as a queue."
active add: (startObject := structure associationAt: object1ID).
done := Set new: structure size.
remaining := structure - (Dictionary with: startObject).
found := false.
"Iteration."
[active isEmpty or: [ found := active contains:
[:vertex| (vertex value contains: [:array | (array at: 1) = object2ID]])]] whileFalse:
[ |currentVertex neighbors |
done add: (currentVertex := active removeFirst).
neighbors := self unprocessedNeighborsOf: currentVertex key
from: remaining.
active addAll: neighbors.
remaining := remaining - (Dictionary withAll: neighbors)].
^found
```

where *neighborsOf: currentVertex* searches collection *remaining* (unprocessed vertices) for the neighbors of the vertex currently being examined. To make its definition more readable, we expressed it using cities instead of general objects:

unprocessedNeighborsOf: aCity from: remainingVertices

"Return collection of associations of all neighbors of aCity that remain unprocessed."

```
| cities unprocessedNeighbors remainingCities neighboringCityArrays |
"Gather names of unprocessed cities that are neighbors of aCity."
remainingCities := remainingVertices keys. "Get names of all remaining city-vertices."
neighboringCityArrays := (structure at: aCity) select: [ "Select those that neighbor aCity."
:setElement | remainingCities contains: [:cityEl | cityEl = (setElement at: 1)]]].
cities := neighboringCityArrays collect: [:array | array at: 1]. "Extract city names out of arrays."
unprocessedNeighbors:= OrderedCollection new.
"Now use cities to get associations – the complete vertex descriptions."
remainingVertices keysAndValuesDo: [:key :value | (cities contains: [:city | city = key])
ifTrue: [unprocessedNeighbors add: (remainingVertices associationAt: key)]]].
^ unprocessedNeighbors
```


Perhaps the trickiest part of these methods was to figure out what the individual messages return. As an example, we made the mistake of using includes: instead of contains: and it took a while before we realized that this was wrong.

To test our code, we used our map and checked whether Montreal is connected to San Francisco (should return true) and whether San Francisco is connected to Toronto (should return false):

```
| cities flights |
"Create data."
cities := Dictionary new
    add: 'Montreal' -> (Set withAll: #(('#('Chicago' nil) #'(New York' nil)));
    add: 'New York' -> (Set withAll: #(('#('San Francisco' nil)));
    add: 'Chicago' -> (Set withAll: #(('#('Denver' nil) #'(New York' nil)));
    add: 'Toronto' -> (Set withAll: #(('#('Vancouver' nil) #'(Montreal' nil) #'(New York' nil)
('Dallas' nil) #'(San Francisco' nil)));
    add: 'Denver' -> (Set withAll: #(('#('New York' nil) #'(San Francisco' nil) #'(Dallas', nil)));
    add: 'Dallas' -> (Set new);
    add: 'San Francisco' -> (Set withAll: #(('#('Dallas' nil) #'(Vancouver' nil)));
    add: 'Vancouver' -> (Set withAll: #(('#('San Francisco' nil) #'(Denver' nil))); yourself.
flights := Graph newWith: cities.
"Do calculations and output results."
Transcript clear
    show: 'Montreal is ',
        ((flights connected: 'Montreal' to: 'San Francisco')
         ifTrue: [''] ifFalse: ['not']),
        ' connected to San Francisco';
cr;
    show: 'San Francisco is ',
        ((flights connected: 'San Francisco' to: 'Toronto')
         ifTrue: [''] ifFalse: ['not']),
        ' connected to Toronto'
```

The test returns the correct result

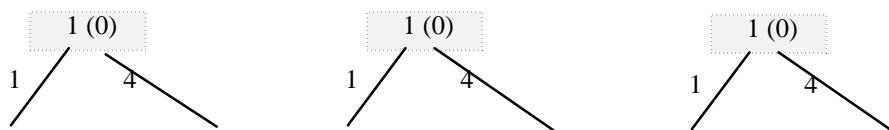
Montreal is connected to San Francisco
 San Francisco is not connected to Toronto

The shortest path in a weighted undirected graph

Consider our road map now and the problem of finding the shortest path from one city to another. Assuming that all weights are non-negative, this can again be done by successive iterations, constantly enlarging the set S of vertices whose shortest distance to the starting vertex v is already known.

At the beginning, we only have one vertex whose shortest distance to v is known - v itself - and its distance to v is 0. We will thus initialize set S to v. In each consecutive step, we examine all vertices that can be reached from any vertex already in S in exactly one step, select the one whose distance to v (calculated from vertices in S) is the shortest, and add it to S. Since this changes S, we recalculate the shortest distance of all vertices not in S that can be reached in one step from S. If the destination node is reachable from v, it will eventually become a member of S and the distance calculated at that point is its shortest distance from v.

This interesting strategy is called a *greedy algorithm* because it always grabs the most appealing choice. We will prove shortly that it indeed gives the desired result but first, let's demonstrate how it works when finding the shortest distance from vertex 1 to vertex 6 in Figure 11.20.



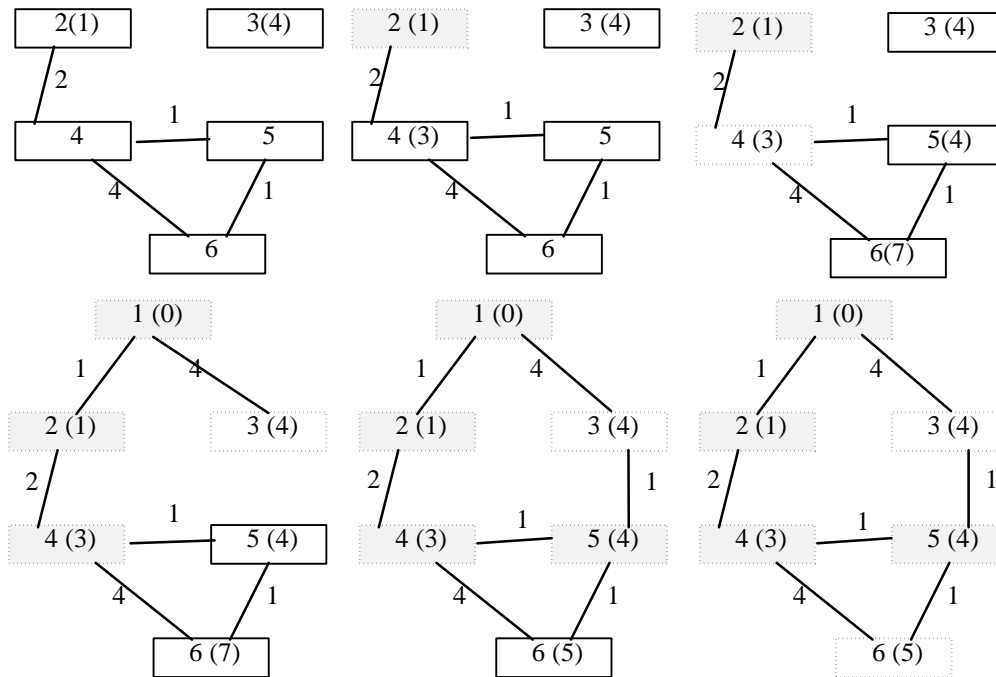


Figure 11.20. Left to right, top to bottom: Finding the shortest distance from vertex 1 to vertex 6.

1. Initialization. Initialize S to source vertex 1: $S = \{1\}$. This is indicated in the top leftmost diagram by showing vertex 1 with a dashed border.
2. Iteration 1. $S = \{1\}$.
 - a. For each vertex reachable from S in one step, calculate the shortest distance from source vertex 1 to this vertex. In our case there are two such vertices - vertex 2 and vertex 3 - and we obtain the distances indicated in the diagram.
 - b. Find the vertex that is not in S and whose calculated distance to vertex 1 is the shortest. In our case, this is vertex 2. Add this vertex to S so that $S = \{1, 2\}$. We indicate that 2 is now in S by drawing its border dashed (second diagram from left).
3. Iteration 2. $S = \{1, 2\}$.
 - a. Recalculate shortest distances to 1 for all vertices not in S . In our case, this does not change existing distances.
 - b. Find the vertex closest to v and not in S . In our case, this is vertex 4. Add this vertex to S (second diagram from left).
4. Iteration 3. $S = \{1, 2, 4\}$.
 - a. Recalculate shortest distances for vertices not in S . No change in existing distances.
 - b. Find the vertex closest to v and not in S . In our case, there are two candidates - vertex 3 and vertex 5, both with distance 4. We arbitrarily choose 3 (third diagram from left).
5. Iteration 4. $S = \{1, 2, 4, 3\}$.
 - a. Recalculate the shortest distances for vertices not in S . No change in existing distances.
 - b. Find the vertex closest to v and not in S and add it to S . This will be vertex 5 (first diagram at left bottom).
6. Iteration 5. $S = \{1, 2, 3, 4, 5\}$.
 - a. Recalculate the shortest distances for vertices not in S . This changes the shortest distance between vertex 1 and vertex 6.

- b. Find the vertex closest to v and not in S and add it to S . This will be vertex 6 (second diagram from left).
7. There is only one vertex left that is not in S - vertex 6. Add it to S and stop (bottom right diagram). The shortest distance from 1 to 6 has now been found and its value is 5. Note that we found not only the shortest distance between 1 and 6 but also the shortest distance between 1 and all other vertices in the graph reachable from vertex 1.

After demonstrating how the algorithm works, we will now prove that it really produces the shortest distance. Our proof is based on induction and indirection.

Proof: Assume that the algorithm works for some intermediate value of S . (It obviously does when S consists of the source vertex only.) Assume that we just added vertex v to S using our algorithm. According to our claim, the distance from the source vertex to v is the shortest distance from the source to v .

Assume, for a moment, that the claim is *false*⁵ and that there is a path giving a *shorter* distance from the source to v . Assume that the first vertex on this alternative path that is outside S is x (Figure 11.21). Let the distance from source to v found by our algorithm be dv , and let the distance from the source to v going through x be dx . If the distance through x is shorter, then $dx < dv$. Since the distance from x to v is not negative,

$$\text{dist}(\text{source} \rightarrow x) \leq \text{dist}(\text{source} \rightarrow x \rightarrow v) = dx < dv$$

This implies $dx < dv$. However, if $dx < dv$, our algorithm would have added x to S rather than v because it always adds the closest reachable vertex. Since it added v , the assumption that the path through x is shorter is false. Consequently, the distance obtained for v by our algorithm is the shortest distance from the source and v .

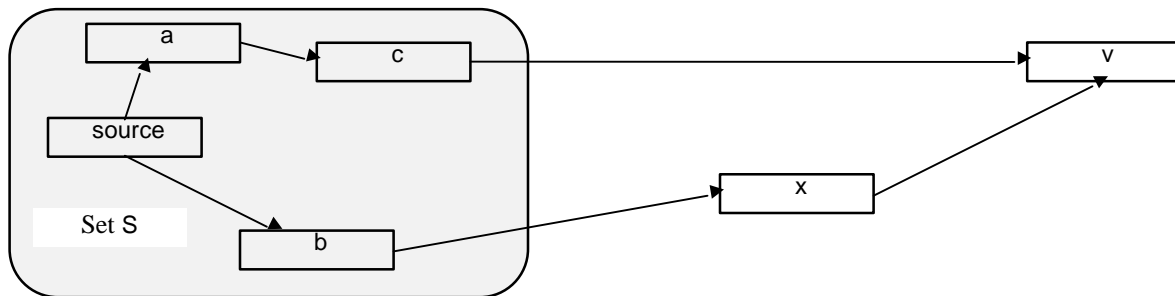


Figure 11.21. Illustration supporting the proof of the shortest path algorithm.

After proving that our algorithm indeed finds the shortest distance, let's formulate it in more detail. Using three collections (remaining, done, activeNeighbors), a more precise description is as follows:

1. Put all vertices except the source vertex s in remaining. Put vertex s in done. Initialize activeNeighbors to an empty collection. Initialize distance of vertex s to 0.
2. Repeat the following until remaining becomes empty or until done contains the destination vertex:
 - a. Move all vertices in remaining reachable in one move from done into activeNeighbors
 - b. For each vertex in activeNeighbors calculate the shortest distance to s via done.
 - c. Move the activeNeighbors vertex whose distance to s is the shortest into done.
3. If done contains the destination vertex, return its distance to s . Otherwise return nil to indicate that there is no path from the source to the destination.

⁵ Proving a claim by showing that its negation is false is called an *indirect proof*.

We leave it to you to implement the algorithm as an exercise.

Main lessons learned:

- A graph consists of nodes (vertices) connected by edges.
- Graphs and operations on them can be very complex and their efficient implementation is one of the major areas of research in Computer Science.

Exercises

1. Extend the flight connection checking method to calculate the smallest number of cities that must be traversed if there is a connection.
2. Prove that our connection checking algorithm indeed fulfills its purpose.
3. Design and implement methods that return the shortest weighted and unweighted path in addition to the weighted or unweighted distance.
4. One of the common applications of graphs is representation of activities that must occur in a certain order. Academic courses and their prerequisites and the sequence of activities in building a house are two typical examples. Sorting a directed graph on the basis of vertex precedences is called topological sort. Formulate, implement, and test an algorithm performing topological sort.
5. Examine our implementation of graph algorithms, find the most inefficient points, and improve the implementation.

Conclusion

This chapter introduced several specialized kinds of collections including stacks, queues, linked lists, trees, and graphs. Although most of them are not explicitly included in VisualWorks library, they are essential for Smalltalk operation and very important in Computer Science applications. In exploring these collections, we introduced several features of internal operation of Smalltalk.

A stack is a last-in first-out structure. Elements are added at the top using the push operation, and removed again from the top using the pop operation. Stack behavior is a part of the behavior of OrderedCollection and there is no need for a Stack class.

An important use of stacks is Smalltalk's execution of messages. Execution of Smalltalk messages depends on a stack of context objects, each of them carrying full information about a message including its receiver and sender, its arguments and local variables, and current state of execution. Each message context also has its evaluation stack for intermediate results. When a message is sent, its context is pushed on the top of the context stack and when finished, the context is popped off. A part of the context is a translation of the code into bytecodes.

Another important example of the use of stacks is exception handling. Smalltalk has a built-in mechanism for dealing with exceptional situations and since this process intervenes into message execution, it is very closely tied to the operation of the context stack. The existence of exception handling allows the programmer to anticipate possible exceptional behaviors and deal with them programatically, preventing the program from raising an exception. Exception handling is achieved by sending a message specifying the desired behavior and the exception handling code to an instance of Signal which then delegates the handling to an instance of Exception. A number of Signal objects for dealing with common exceptions are built into the library and users can define their own as well.

A queue is a first-in first-out structure where elements are added at one end and removed from the other. Queue behavior is subsumed by OrderedCollection and there is no need for class Queue. One of the most important applications of queues is in simulation but the Smalltalk run-time environment also uses queues for several operations. Some of these will be covered in Chapter 12.

A list is a linear collection in which each element knows about its successor (single linked list) or its successor and predecessor (doubly linked list). VisualWorks library contains a pair of general classes

called `LinkedList` and `Link` implementing the basic linked list behavior. For concrete use, these classes are usually subclassed. The advantage of links is that they allow easy insertion and deletion.

A tree is a branching structure of nodes and their children. The node at the top of a tree is called the root. Every node in a tree except the root has exactly one parent. The root does not have a parent. The bottom nodes in a tree - the nodes that don't have any children - are called leafs.

In general, a node in a tree may have any number of children but specialized trees may restrict the number of children a node is allowed to have. As an example, a node in a binary tree may have at most two children. A very important use of trees is in compilation but the Smalltalk compiler does not build a tree explicitly. Instead, it constructs a nested structure of node objects equivalent to a tree.

Graphs are the most complex type of collection. A graph consists of nodes (vertices) connected by edges. Edges may be directed or undirected, weighted or unweighted. Graphs and operations on them can be very complex and their efficient implementation is one of the major areas of research in Computer Science. Since the operation of Smalltalk does not require any graphs, graphs are not included in the library.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

`CompiledMethod`, **Exception**, **Signal**, `LinkedList`, `Link`.

Terms introduced in this chapter

binary tree - a tree allowing at most two children per node

breadth-first algorithm - an algorithm that deals with all children of a node before examining the children's children

context - information needed to execute a message including its code, sender, receiver, arguments, temporary variables, current state of execution, and working stack

context stack - stack of contexts of all currently active messages stored in order of execution

depth-first algorithm - an algorithm that follows a complete path to a leaf before dealing with sibling nodes on the same level

exception - abnormal behavior such as attempt to divide by zero or attempt to access an illegal index

exception handling - execution of predefined code when an exception occurs

graph - a collection of vertices connected by edges, possibly directed and weighted

leaf - a tree node with no children

lexical analysis - the process of converting textual source code into a collection of program components such as number, word, binary selector, or keyword; first step in compilation

linked list - linear collection held together by single links

parsing - the process of recognizing grammatical constructs such as statements and blocks during compilation; follows scanning and precedes code generation

pop - the act of removing an element from the top of a stack

push - the act of adding an element at the top of a stack

queue - linear collection where elements are added at one end and removed at the other

root - the top node of a tree; has no parent

scanning - synonym of lexical analysis

stack - linear collection where elements are added and removed at the same end

tree - collection of object nodes in which each node may have one or more children, and where each node except the root has exactly one parent

Chapter 12 - Developing user interfaces

Overview

The group of classes supporting graphical user interfaces (GUIs) is one of the most complex parts of VisualWorks and its detailed coverage is beyond the scope of this book. Fortunately, to be able to use the tools for routine applications and to create interesting extensions of the widget library, you only need to understand a limited number of concepts, and specialized books are available for more advanced uses.

The first requirement for creating a user interface is to be able to draw objects such as lines, widgets, and text on the screen. VisualWorks perspective is that drawing requires a surface to draw on (such as a computer screen or printer paper), the graphical objects being drawn, and an object that does the drawing and holds the parameters that describe the drawing context (such as fonts, colors, and line widths).

Being able to draw an object on the screen is, however, only a part of a typical GUI. Unless the displayed windows are completely passive, the user must also interact with them via the mouse and the keyboard. A user interface must also ensure that the display reflects changes of the displayed data model, and that damage caused, for example, by resizing a window is automatically repaired.

To implement automatic response to model changes and user interaction, Smalltalk uses the model-view-controller paradigm - MVC for short. The model part of the MVC triad holds data and provides notification when the data changes. The view is responsible for drawing the component on the screen, response to model changes, and damage repair. The controller handles user input.

To maintain the UI painting paradigm supported by the UI Painter, we also need a way to integrate a new visual component into a canvas. This is achieved by a widget called the subview which implements access to its view part and the connection to its window container.

12.1 Principles of user interfaces - display surfaces, graphics contexts, and visual parts

This section is dedicated to the principles of VisualWorks user interfaces and provides background for customization of windows and their main components. In routine applications, no programming involving these components is necessary but understanding of these concepts is useful.

Displaying graphical objects on the screen is a complex task and different languages and even different dialogs of Smalltalk approach it differently. The view of VisualWorks is that drawing requires a drawing surface, objects to be displayed, and an object that can do the drawing and holds parameters describing how and where the drawing takes place. We will now deal with these concepts one by one and illustrate them on simple examples. A larger example follows in the next section.

Display surfaces

Display surfaces are the media on which all visuals, including geometric objects, images, widgets, and text, are displayed. The hierarchy of display media is large and its skeleton is as follows:

```
Object
  GraphicsMedium
    DisplaySurface
      UnmappableSurface
        Mask
        Pixmap
      Window
        ScheduledWindow
          ApplicationWindow
            TransientWindow
      HostPrintJob
      MockMedium
      PostScriptFile
```

The abstract class `GraphicsMedium` at the top of the hierarchy defines the protocol for accessing information such as display defaults (paint, font, and look preferences which include background and foreground color), size of the display surface, number of bits used to represent the color of pixels, and graphics context (an object responsible for display of geometric objects, images, and text). `GraphicsMedium` All the supporting instance variables are defined in subclasses.

Subclasses of `GraphicsMedium` can be divided into those displaying on the screen, those used to construct displayable surfaces in memory, and those dedicated to printing. We will deal only with the first two categories and leave printing to you as an exercise. The group of classes responsible for screen display and display construction are headed by `DisplaySurface` whose comment essentially says

Class `DisplaySurface` is the abstract class for all classes that act as destinations for graphics operations targeted for a bitmap video display (as opposed to printing, for example).

Instance variables:

handle	<GraphicsHandle nil>	handle to the host resource, an interface to the operating system which performs the low level display functions
width	<SmallInteger>	width of the surface in pixels
height	<SmallInteger>	height of the surface in pixels
background	<Paint>	paint used to clear the surface

Even though `DisplaySurface` is abstract, it provides a lot of useful functionality such as closing the display surface (for example, a window) and access to any part of the display area. Subclasses of `DisplaySurface` can be subdivided into classes whose instances correspond to windows displayed on the screen, and classes that instantiate display surfaces in memory but don't display them.

Windows

Class `Window` at the top of the window hierarchy is a concrete class with much functionality but rather useless by itself because its instances have neither a controller (thus allowing only minimal user interaction) nor a component (thus incapable of containing widgets). A lot of the behavior inherited from `Window` is, however, essential. This includes the knowledge of the window's origin on the screen, its label, its icon, and its sensor which holds information about mouse and keyboard events that occur within the window's extent. `Window` also defines methods for opening a window (redefined in subclasses), collapsing it into an icon and re-expanding it, moving and resizing a window, and raising it to the top to become the active window on the screen. It also knows the currently active window (class message `currentWindow`).

As an example of window manipulation, execute the following code fragment to see some of the `Window` functionality:

```
"Open window, collapse it, expand it, and close it."  
| window |  
window := Window openNewIn: (100@100 corner: 200@200).  
(Delay forSeconds: 3) wait.  
window label: 'Test'.  
(Delay forSeconds: 3) wait.  
window collapse.  
(Delay forSeconds: 3) wait.  
window expand.  
(Delay forSeconds: 3) wait.  
window close    "Without this statement, you will not be able to close the window."
```

Class `Window` has very little direct use and we will thus proceed to its subclass `ScheduledWindow`. This class adds several behaviors and instance variables, the most important being controller (handles user interaction), component (makes it possible to include a visual part or a collection of visual parts in the window), and model (allowing another object to control a window). The controller of `ScheduledWindow` is an instance of `StandardSystemController` which provides the <window> menu and methods for closing,

resizing, moving, relabeling, and other window operations. The window controller is also the means by which Smalltalk keeps track of all its windows and the controllers of all open Smalltalk windows are kept in `ScheduledControllers`, an instance of `ControlManager`. The following is a simple example showing how to open a `ScheduledWindow`:

```
"Open window with label but with no component, and let user close it."  
| window |  
window := ScheduledWindow new.  
window insideColor: ColorValue blue;  
        label: 'Test'.  
window openIn: (100@100 corner: 200@200)
```

The user can close a `ScheduledWindow` in the usual way which is not true for a `Window`. The window in this example does not have any components so it's not of any interest but we will do an example with a component later.

Until recent VisualWorks releases, `ScheduledWindow` was the basis of all user interfaces - there was no UI Painter and all UIs had to be created programmatically using `ScheduledWindow`. With the introduction of the UI Painter, this has changed and applications based on the application model use the `ApplicationWindow` subclass of `ScheduledWindow`. `ScheduledWindow` has thus lost its importance except that it defines many behaviors inherited by `ApplicationWindow`. In the current class library, most references to `ScheduledWindow` are in examples because (as our example shows) it is easier to create a very simple demo window programatically than to paint one using the UI Painter.

From within an application, the `ApplicationWindow` of the interface can be accessed by sending message `window` to the application builder. As an example, the application model could send

```
self builder window label: 'New label'
```

to change the window's label at run time. The essential part of the comment of `ApplicationWindow` is as follows:

`ApplicationWindow` adds functionality required by the User Interface building classes. It also provides for tighter coordination with a corresponding `ApplicationModel`. In particular, an `ApplicationWindow` can notify its application of window events (e.g. closing or collapsing), so that other dependent windows can follow suit. Recognized events are `#expand`, `#collapse`, `#bounds`, `#enter`, `#exit`, `#close`, `#hibernate`, `#reopen`, and `#release`.

Instance Variables:

<code>keyboardProcessor</code>	<KeyboardProcessor>	Manages keyboard input
<code>application</code>	<Object nil>	The application, typically an <code>ApplicationModel</code>
<code>receiveWindowEvents</code>	<Array nil>	Window events the receiver should respond to
<code>damageRepairsLazy</code>	<Boolean>	Used to control when damage coming up from below is repaired.

Variable `damageRepairsLazy` is used to determine whether a damaged window should be repaired (redrawn) immediately or whether redisplay can wait until a suitable later time. We will talk about damage repair later.

Typical messages sent by the application model to an application include changing its label or background color at run time (messages `label:` and `background:`), moving it to a new position (message `moveTo:`) obtaining its bounding rectangle (`displayBox`), closing the window (`close` - usually achieved by sending `closeRequest` to the application model), collapsing it to an icon and expanding the icon back to a window (`collapse` and `expand`), and hiding the window without destroying its internal representation (message `unmap`) and restoring it without having to reconstruct it (message `map`). In many cases, a window changed at run time must be redisplayed by message `display`.

When an application model needs to access more than one window, create a new builder for each of the secondary windows, specify its source as the application model, specify the window's application as the application model, and ask the secondary builder to open the interface. You can also make the secondary window a slave of the master window (message `slave`). The result is that the slave will mimic closing, collapsing, and expanding of the master window. The slave window may also be made sensitive to

other master window events such as cursor entry or exit. Finally, windows can also be partners by sending `bePartner` to the window. Partner windows are tied together in the same way as master and slave windows.

We will see examples of some of this functionality in the following sections and other examples are provided in the cookbook.

Main lessons learned:

- In the perspective of VisualWorks, display requires a display surface, objects to be displayed, and a drawing engine (a graphics context).
- Display surfaces are subclasses of `GraphicsMedium` which has two subtrees, one for display on screens and one for printing on paper.
- Display on screen uses windows and the window hierarchy includes classes `Window`, `ScheduledWindow`, and `ApplicationWindow`.
- Windows created with the UI Painter and used by application models are `ApplicationWindows`. They inherit most of their functionality from `ApplicationWindow` superclasses.
- `ApplicationWindow` has numerous instance variables holding parameters such as bounding box, clipping rectangle, graphics context, background paint, and others.
- `Pixmap` and `Mask` are display surfaces used to construct displays before they are mapped to the screen.

12.2 An example of the use of windows – a Virtual Desktop

One annoying aspect of graphical user interfaces is that computer screens are always too small to display all the windows that one would like to see at a time. A partial solution to this problem is a ‘virtual desk’ program which allows the user to place individual open windows on ‘desks’ and view only one desk at a time. We will design and implement a very simple version of such a program. We will call it `Desk Manager` and implement it via class `SmallDesk`.

Our program will make the screen behave as a peep hole on a much larger desktop divided into four screen-size rectangular areas. The selection of the desired desk is via the user interface in Figure 12.1. The four buttons on the left represent the virtual desks and the button of the currently displayed desk is highlighted. When the user clicks an un-selected desk, `SmallDesk` hides all VisualWorks windows on the current desk and displays all VisualWorks windows assigned by the user to the selected desk. The Desktop window itself remains visible at all times.

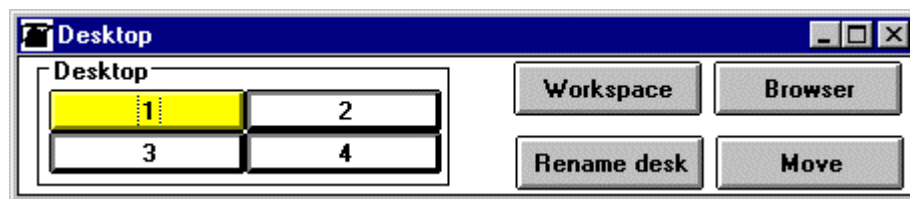


Figure 12.1. Default user interface of `SmallDesk` manager.

The buttons on the top right allow the user to open a `Workspace` and a `Browser` on the current desk because opening a `Browser` or a `Workspace` is the most common way to start populating a desk with windows. The two buttons on the bottom right allow the user to rename desk buttons and to move a selected window from one desk to another. A desktop can also be opened with other than default desk names and the user can specify which applications should be displayed on individual desks. Additional specification details:

- When `SmallDesk` opens with the default open message, it selects desk 1 and maps (displays) all windows already on the screen on this desk. All other desks are empty.

- When SmallDesk closes, it maps all windows from all desks. In other words, it displays all VisualWorks windows assigned to all four desks on the screen.
- When the user clicks *Rename desk*, a prompt requests the new name for the current desk and displays it on the button.
- The *Move* button allows the user to move a window from the current desk to another desk. It opens two consecutive prompts that allow the user to identify the window to be moved and its new desk. If the destination is different from the current desk, the selected window is unmapped and assigned to the specified desk; no desk switch occurs. The Desk Manager window itself cannot be moved to another desk and is always displayed.
- The user can also open SmallDesk by one of several more powerful messages. As an example, to open the desk on desk 2 with button names, and assign application LecturePad to the first desk, two workspaces to the second, a Browser to the third, and two copies of Notes to the fourth, the user will execute

SmallDesk

```
openWithManyApps: #(LecturePad) #(ComposedTextView ComposedTextView) #( Browser)
                  #(Notes Notes))
andNames: #('desk 1' 'desk 2' 'desk 3' 'desk 4')
onDesk: 2
```

Preliminary design

We only need one class - an application model to be called **SmallDesk**. This class will be responsible for user interface specification, opening messages, and action buttons. We will keep the mapping of windows to desks in a dictionary, using windows as keys and integer numbers of desks as values.

The major *responsibilities* of **SmallDesk** are as follows:

Opening. The default opening message assigns all VisualWorks windows on the screen to desk 1 and highlights its button. Specialized opening messages do one or more of the following: rename buttons, assign specified applications to specified desks, open on specified desk.

Action methods. Action methods can be divided into the following groups:

- *Desk button* action methods. The user clicks a desk button either to switch to a different desk or to identify the destination desk when moving a window from one desk to another.
- *Browser* button and *Workspace* button open the Browser and the Workspace on the current desk.
- The *Rename* button displays a prompt and assigns the obtained string as the label to the currently selected desk button.
- The *Move* button displays a prompt to identify a window, another prompt to identify the destination desk, unmaps the window from the current desk, and assigns it to the destination map. An attempt to move the Desk Manager window itself will be refused with a warning.

Closing. This button will prompt the user to confirm when he or she attempts to close SmallDesk. It will then map all windows to the screen and close SmallDesk.

Design refinement

Before writing the specification of **SmallDesk**, we will examine some of the details that we ignored in preliminary design starting with instance variables. We need a window registry that holds information about the assignment of windows to desks. We also need to know the current desk's number so that we can rename it and map and unmap windows on desk change request, and check whether a new desk selection is different from the current one.

To deal with the two different modes of operation that affect the result of clicking a desk button (switching to a different desk versus renaming it), we must also keep track of the mode. We will represent it by a **Symbol** which will be identical to the name of the method that executes the operation so that we can execute the method by the **perform:** message. Finally, we will use 'lazy opening' to open applications specified by the user in the opening message. **SmallDesk** will initially open only applications on the opening desk, and the remaining applications will only be opened when the user switches to the desk assigned to them. We will keep unopened applications in a four-element array with one element corresponding to each desk.

We can now start thinking about the methods. Default initialization puts all **VisualWorks** windows on the screen on desk 1, sets current desk number to 1, and mode to 'switch to another desk'. It also initializes the array of unopened applications.

The method performing desk change checks whether the new desk is different from the current one (and do nothing if it is), change the colors of backgrounds of the current and new desk buttons, update window registry (windows may have opened or closed on the current desk since the last registry refresh and this must be recorded), unmap the windows on the current desk, map windows on the new desk, open the unopened applications assigned to this desk, and make new desk the current desk.

Finally, when the user clicks to close **SmallDesk** and confirms, all scheduled windows from all desks are mapped to the screen.

We can now write the following detailed specification of class **SmallDesk**:

SmallDesk. This class helps the user to reduce screen clutter by creating four virtual screens called desks. **SmallDesk** continuously displays a Desk Manager window with four buttons representing the virtual desks and allows **VisualWorks** windows to be associated with any one of them. Only windows assigned to the current desk are displayed. Desk buttons can be renamed and windows can be moved from one desk to another. The application can open on any desk, with any combination of desk button names, and with specified applications assigned to individual desks.

Superclass: **ApplicationWindow**.

Instance variables: **windowRegistry** <**IdentityDictionary**> pairs windows with desktops (integers), **waitingApps** <**Array**> contains an array with class names of unopened applications for each desk, **currentDesk** <**Integer**> identifies currently displayed desk, **mode** <**Symbol**> identifies the current context of operation - one of **#changeTo:**, **#moveTo:**

Responsibilities:

- Opening
 - **openWithManyApps:** and **Names:** **onDesk:** opens the application on the specified desk and with the specified applications. All currently displayed Smalltalk windows are assigned to desk 1.
 - **open** – special case of the above method that opens with default button names, on desk 1, and with no new applications.
- Initialization and closing.
 - **initialize** – the standard opening hook method. Gets all **VisualWorks** windows from **ScheduledController** to initialize **windowRegistry**, initializes **currentDesk** to 1 (overridden later when a different desk is specified), **mode** to **#changeTo:**.
 - **closeRequest** asks the user to confirm **SmallDesk** closing, maps all windows referenced by **ScheduledControllers** and allows the Desk Manager window to close.
- Action methods responding to UI buttons.
 - **desk1**, **desk2**, **desk3**, **desk4** methods all act in the same way and the only difference between them is that each identifies a different desk. Consequently, they all send private message **updateDesk:** with their number as the argument and further processing happens in this method. Since the same method is used for changes, its operation depends on the context: It performs one of the two possible actions corresponding to the two possible contexts in which the button may be activated. These two actions will be implemented by
 - **changeTo:** - checks whether the new desk is different and if it is, hides currently displayed windows, displays windows assigned to the new desk, opens waiting applications assigned to this desk, and colors the buttons appropriately.

- `moveTo:` is triggered by the *Move* button via the `move` method (below). It displays prompts to select the window to be moved and the destination desk, unmaps the window, and assigns the number of the specified desk to this window in the registry.
- `openBrowser` and `openWorkspace` buttons open the Browser and the Workspace.
- `rename` requests a new name for the currently selected desk button and assigns it to the button as its label.
- `move` asks the user to select a window and click the destination desk button. The internal assignment of the desk is performed by `updateDesk:`.
- Private support methods.
 - `updateDesk:` is activated by a desk button. It performs the value of the `mode` variable and supplies the desk number as the argument. The result is either a change of desk (method `changeTo:`) or reassignment of a window to a new desk (method `moveTo:`), depending on the context.

Implementation

We will now list and explain some of the methods.

Opening

openWithManyApps: classArray andNames: nameArray onDesk: anInteger

"Open on desk anInteger, with classes as specified in classArray, and button names according to nameArray. Applications that are not on desk anInteger do not open yet."

| desk deskBuilder |

desk := self new.

"Create the desk manager but stop short of opening so that the manager window can be positioned, etc."

deskBuilder := desk allButOpenInterface: #windowSpec.

"Rename desk buttons via the builder and button widget ids."

1 to: 4 do: [:index | | buttonView |

buttonView := (deskBuilder componentAt: ('desk' , index printString) asSymbol) widget.

buttonView label: (Label with: (nameArray at: index)).

"Position Desk Manager window in upper left."

deskBuilder source finallyOpenIn: (10 @ 10 extent: deskBuilder window bounds extent) withType: #normal.

"Assign user specified applications to desks."

1 to: 4 do: [:index | | classes |

((classes := classArray at: index) isNil or: [classes isEmpty])

ifFalse: [classes do:

[:class | desk waitingApps at: (Smalltalk at: class) put: index]]].

"Open user-specified desk."

desk openDesk: anInteger.

^deskBuilder

The opening of the desk is executed by

openDesk: integer

"Switch to the specified desk or edit button label or move window."

self perform: mode with: integer

A part of the execution of `new` is execution of the hook method `initialize` which is defined as follows:

initialize

"Initialize registry of windows and desks, start in 'move' mode with existing windows in desk 1."

waitingApps := Array with: Set new

with: Set new

with: Set new

with: Set new.

windowRegistry := IdentityDictionary new.

ScheduledControllers scheduledControllers do: [:controller | windowRegistry at: controller view put: 1].

```
self changeSystemController.  
mode := #changeTo:
```

The response to a desk button clicks is handled by methods desk1, desk2, desk3, and desk4 defined as follows:

desk1

```
"Action method of button desk 1."  
self updateDesk: 1
```

The operation of updateDesk: depends on the current mode and its definition is as follows:

updateDesk: integer

```
"Depending on the context, switch to integer desk or edit button label or move window."  
self perform: mode with: integer
```

Depending on the value of mode, this method performs one of changeTo: or moveTo:. Method changeTo: switches from one desk to another and its definition is

changeTo: newDesk

```
"Switch to the specified desk if different from the current one."  
newDesk = currentDesk ifTrue: [^nil].  
"Change button colors."  
currentDesk isNil ifFalse: [self colorButton: currentDesk color: ColorValue white]. self colorButton: newDesk  
color: ColorValue yellow.  
"Update the registry – windows may have been added or removed."  
self refreshWindowRegistry.  
currentDesk := newDesk.  
"Go through registry, mapping windows on the current desk and unmapping the rest. Always keep the Desk  
Manager – the window corresponding to self."  
windowRegistry keysAndValuesDo: [:key :value | (key = self builder window or: [value = newDesk])  
ifTrue: [key = self builder window ifFalse: [key map]]  
ifFalse: [key unmap]].  
"Open unopened applications waiting on this desk, if any."  
(waitingApps at: currentDesk) do: [:app | self open: app]
```

An important part of the definition is method refreshWindowRegistry which is defined as follows:

refreshWindowRegistry

```
"We are switching to a different desk. Make sure to update the desk being abandoned - windows may have  
been added or deleted."  
| newRegistry |  
"Create a new empty registry ."  
newRegistry := IdentityDictionary new.  
"Copy active windows in the registry and new windows to the new registry."  
ScheduledControllers scheduledControllers do:  
    [:controller |  
    | window |  
    (windowRegistry includesKey: (window := controller view))  
    ifTrue: [newRegistry at: window put: (windowRegistry at: window)]  
    ifFalse: [newRegistry at: window put: currentDesk]].  
"Replace old registry with new one."  
windowRegistry := newRegistry
```

The open: method on the last line of changeTo: opens a waiting application, removes it from its array and adds the resulting window to the registry:

open: anApplication

```
"Open anApplication, remove it from waitingApps, and add its window to the registry."
```

```
| window |  
(waitingApps at: currentDesk) remove: anApplication.  
window := anApplication open.  
windowRegistry at: window put: currentDesk
```

Note that we assume that all unopened applications open with the open message. This restriction could, of course, be easily removed. The moveTo: method moves a selected window to a selected desk. Its definition is

moveTo: newDesk

```
"Move previously selected window to the specified desk (registry) and hide it."  
newDesk = currentDesk ifTrue: [^nil].  
windowRegistry at: currentWindow put: newDesk.  
currentWindow unmap.  
mode := #changeTo:
```

Note that the method ends by switching to the default mode of operation.

Who sends the moveTo: message? Its trigger is the activation of the *Move* button whose action method move first lets the user select a window by waiting until ScheduledControllers reports that the user pressed the red button, and then identifies the active window. After this, the user is asked to click a desk button (to identify the destination desk). The mode changes to #moveTo:, and when the user clicks one of the desk buttons, this sends updateDesk: (as listed above), which then performs moveTo:. The action method move is defined as follows:

move

```
"User clicked Move to move a window to a different desk. Identify the window and ask user to select  
desk. The rest of the operation is handled in moveTo:."  
Dialog warn: 'Select window to be moved'.  
"Wait for user to press leftmost mouse button."  
[ScheduledControllers activeController sensor redButtonPressed]  
whileFalse: [].  
"Get active window, assuming that this is the one the user wants."  
currentWindow := Window currentWindow.  
"Don't let the user try to move the Desk Manager window to a different desk."  
currentWindow = self builder window ifTrue: [^Dialog warn: 'You cannot move Desk Manager window'].  
Dialog warn: 'Click destination desk'.  
mode := #moveTo:
```

Finally, opening the browser and the workspace is done as follows:

openBrowser

```
"Open Browser on current desk."  
(Smalltalk at: #Browser) open
```

and

openWorkspace

```
"Open Workspace on current desk."  
(Smalltalk at: #ComposedTextView) open
```

Exercises

1. Design and write a method to open SmallDesk with applications whose opening message is not open.
2. An alternative to unmapping windows is to collapse them to icons. Change SmallDesk to use collapsing.
3. SmallDesk mechanism for moving a window to another desk is awkward: The user must click the *Move* button, complete the dialog, click a window, complete another dialog, and finally click the destination desk. To simplify the process, add a new '*move to another desk*' command to the <window> menu

when SmallDesk is running. When the user selects this command, SmallDesk asks him or her to click a desk button, and when the user does so, SmallDesk moves the window to that desk.

12.3 Principles of displaying - graphics contexts, geometric objects, and other concepts

In this section, we will explain the principles of displaying components inside a window and illustrate them on simple examples.

Graphics context

When we want to display a geometric object such as a rectangle, we must specify which part of the display surface we want to paint on, and which color and line width to use. For text, we must also know the font. VisualWorks keeps this information in a GraphicsContext object attached to the display surface. The GraphicsContext knows about its graphics medium and is also responsible for all display operations. We will see numerous examples of the use of GraphicsContext but first, we will examine its definition. The place of GraphicsContext in the class hierarchy is as follows:

Object

```
GraphicsContext
  HostPrinterGraphicsContext
  PostScriptGraphicsContext
  ScreenGraphicsContext
```

and the essence of its comment says

I display graphical objects on a display medium and maintain graphics state parameters specifying how graphical objects should be rendered.

Instance variables:

medium	<GraphicsMedium>	where to render graphics
clipOriginX	<nil SmallInteger>	in untranslated device coordinates
clipOriginY	<nil SmallInteger>	In untranslated device coordinates
clipWidth	<nil SmallInteger>	
clipHeight	<nil SmallInteger>	
lineWidth	<SmallInteger>	draw lines with this width in pixels
capStyle	<SmallInteger>	style to render line caps
offsetX	<SmallInteger>	translation
offsetY	<SmallInteger>	translation
phaseX	<SmallInteger>	tile phase for tiling the background
phaseY	<SmallInteger>	tile phase
scaleX	<Float>	scaling factor
scaleY	<Float>	scaling factor
font	<ImplementationFont>	font for drawing strings
paint	<Paint>	color or tile for displaying graphical entities
paintPolicy	<PaintPolicy>	how to render paints that do not exactly correspond to host paints
fontPolicy	<FontPolicy>	how to map FontDescriptions to the fonts available on the graphics device

A more detailed description of some of the less obvious instance variables is as follows:

clipOriginX, clipOriginY, clipWidth, clipHeight describe the clipping rectangle - the part of the display surface on which the GraphicsContext will draw. This rectangle acts as a stencil and all drawing that falls outside it is ignored.

capStyle specifies the style used for straight line endings; it is visible only on thick lines.

offsetX, offsetY: translates coordinates of graphics context with respect to the origin of its medium

scaleX, scaleY: scaling factor

paint: Default paint used when the visual part being drawn does not specify its paint. The concept of a paint requires a brief comment. Display surfaces can be painted either with colors or with patterns. The distinction is similar to the distinction between painting a wall and wallpapering it: A color paints the whole object the same color (filling a rectangle with red color, for example) whereas pattern repeats a tiling pattern across the whole area.

phaseX, phaseY: Tile phase specifies the starting point of the first tile of a painting pattern.

paintPolicy, fontPolicy: If the platform cannot provide the required color or font, this object finds a suitable available replacement.

Most display methods are defined in this class, and subclasses implement differences between screen display and printing such as conversion between pixels (screen distance units) and printer units (corresponding but different printer distance units).

We will now present two examples showing how to use the `GraphicsContext` to display text and geometric objects in a window, how to control display via the graphics context, and how to use patterns. You will note that our code does not provide automatic damage repair which means that if you change window size, collapse and expand it, or partially cover the window and then uncover it, the drawing will be lost. As we noted earlier, automatic damage repair is a separate mechanism that will be covered later.

Example 1: Drawing geometric objects and controlling their color

Problem: Implement an application with the user interface in Figure 12.2. Clicking *Color* opens a multiple choice dialog listing all available colors and accepts the selected color for drawing. Clicking *Draw* opens a dialog requesting information needed to display the object selected by the radio button, for example the end points of a straight line when *Line* is selected. A predefined clipping rectangle inside the window is used to restrict display to the upper part of the window.

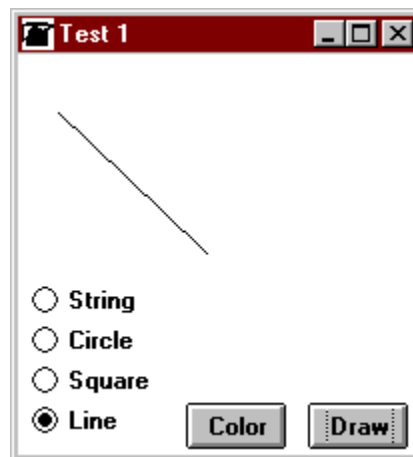


Figure 12.2. Example 1: Clipped drawing of a line with end points in the upper left and lower right corners of the application window.

Solution: The solution revolves around the graphics context which controls the parameters and does the drawing. The graphics context is obtained from the window which, in turn, is known to the builder of the application. From within the application model, the graphics context of the window is thus accessed by

builder window `graphicsContext`

Although this expression suggests that a window keeps its graphics context, this is not so. Instead, a window manufactures a graphics context with default parameters whenever it is asked for one. Changes made to a graphics context are thus transient, and when a graphics context is modified and then retrieved

another time, it will again be a graphics context with default parameters. This is an important but often forgotten principle.

To implement the program, we painted the user interface installed it on class `DrawObjects`, and defined instance variables `color`, `clippingRectangle`, `object`, and `width` and initialized them as follows:

initialize

```
"Set initial parameters for use on graphics context."  
  clippingRectangle := 20 @ 20 corner: 180 @ 100.    "Upper part of the window."  
  color := ColorValue black.                          "Default color."  
  object := #Line asValue.                            "Aspect variable of the radio button group."  
  width := 1
```

The `color` action method gets a color `Symbol` from a dialog and executes it to get a `ColorValue`:

color

```
"Get color from user and save it."  
| colorName |  
colorName := Dialog      choose: 'Choose color'  
                        fromList: ColorValue constantNames    "Names of built-in named colors."  
                        values: ColorValue constantNames  
                        lines: 8  
                        cancel: [nil].  
colorName isNil ifFalse: [color := ColorValue perform: colorName]
```

The action method of the *Draw* button obtains the window's graphics context, changes its paint, line width, and clipping rectangle according to instance variables, and sends a message to draw the selected object:

draw

```
"Get graphics context, set its parameters, and execute appropriate drawing message."  
| gc |  
gc := self builder window graphicsContext.  
gc paint: color.  
gc lineWidth: width; clippingRectangle: clippingRectangle.  
self perform: 'draw', object value, 'On:' with: gc
```

The last statement is possible because we assigned radio buttons aspects to match the required display message, such as `#Circle`. As a consequence, when the *Circle* button is selected, the last line is equivalent to `self drawCircleOn: gc`. Each of the drawing methods then obtains the appropriate parameters from the user and asks the graphics context to draw the object. As an example, the method to draw a square is defined as follows:

drawSquareOn: aGraphicsContext

```
"Get origin coordinates and side length from user, draw a square."  
| startx starty side |  
'Obtain origin and side of square from user.'  
startx := (Dialog request: 'Upper left x' initialAnswer: '30') asNumber.  
starty := (Dialog request: 'Upper left y' initialAnswer: '30') asNumber.  
side := (Dialog request: 'Side' initialAnswer: '20') asNumber.  
"Draw unfilled square inside the clipping rectangle using current parameter settings."  
^aGraphicsContext displayRectangularBorder: (startx @ starty extent: side @ side)
```

It is worth noting that instead of asking a graphics context to draw a visual object, we can also ask the object to draw itself using the graphics context. As an example, the effects of

```
aGraphicsContext displayRectangularBorder: aRectangle
```

and

aRectangle displayOn: aGraphicsContext

are exactly the same. The advantage of the second method is that it can be used even if we don't know the kind of visual part being drawn, as when we have a whole collection of objects to draw:

objects do: [:object| displayOn: aGraphicsContext]

We will use this approach in another example.

Example 2: Drawing with patterns and drawing with color

Problem: Design an application with the user interface Figure 12.3. When the user selects a color via the *Color* button and clicks *Draw*, the program requests square parameters from the user and draws a square filled with the selected color. When the user clicks *Pattern*, the program displays the cross hair cursor and the user selects a rectangle on the screen. If the user then clicks *Draw*, the program requests square parameters and tiles the square with copies of the selected area.

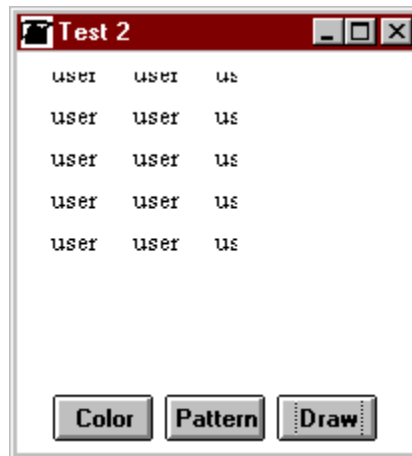


Figure 12.3. Example 2: Displaying a square filled with a pattern copied from the screen.

Solution: In class *Patterns*, we define instance variable *paint*, and initialize it to black color:

initialize

```
paint := ColorValue black
```

The color method is as in Example 1 and the action method of the *Pattern* button is as follows:

pattern

```
"Ask user to select screen area with desired pattern."  
Dialog warn: 'Select a rectangle on the screen'.  
paint := Screen default contentsFromUser asPattern
```

The action method of the *Draw* button obtains a rectangle from the user, assigns the paint to the graphics context, and asks the graphics context to draw the square with the assigned fill:

draw

```
"Obtain square parameters from user and draw the square filled with the current paint."  
| startx starty side gc |  
"Get parameters."  
gc := self builder window graphicsContext.
```

```
gc paint: paint.  
startx := (Dialog request: 'Upper left x' initialAnswer: '30') asNumber.  
starty := (Dialog request: 'Upper left y' initialAnswer: '30') asNumber.  
side := (Dialog request: 'Side' initialAnswer: '20') asNumber.  
"Disply."  
^gc displayRectangle: (startx @ starty extent: side @ side)
```

Main lessons learned:

- All display within a window is performed by an instance of GraphicsContext.
- GraphicsContext is central to everything related to display of visual components.
- In addition to being the drawing engine, a GraphicsContext also holds display parameters such as line width, paint, and fault.
- Display parameters stored in the graphics context can be controlled programmatically.

Exercises

1. Browse and explain the operation of GraphicsContext.
2. Browse and explain class Paint.
3. Modify Example 1 to obtain the clipping rectangle from the user rather than using a fixed preprogrammed value. (Hint: Look for implementers of method fromUser:.)
4. Modify Example 2 to ask the user to specify a rectangle interactively instead of obtaining square parameters via a dialog. (Hint: Look for implementers of method fromUser:.)
5. We mentioned that asking an object to draw itself on a graphics context is equivalent to asking the graphics context to draw the object. This suggests that one of the two approaches is implemented by double dispatching. Check whether this is indeed the case.

12.4 Images, pixmaps, masks, and paints

In addition to displaying geometric objects and text, we can also display images and use masks to control which part of the image is visible. In this section, we will explain the principles of these concepts and illustrate them on simple examples. We will also talk about cursors because they are related but we will leave an example of their use for later.

Image

An Image (more accurately an instance of one of its concrete subclasses) is a rectangular array of pixels, each represented internally by a group of bits and displayed as a colored dot. Since different images may use a different number of bits to represent pixel color, a single class cannot represent all possible kinds of images. Image is thus an abstract class which leaves concrete image representation to its subclasses:

Image

- Depth1Image
- Depth2Image
- Depth4Image
- Depth8Image
- Depth16Image
- Depth24Image
- Depth32Image

The depth of an image is the number of bits used to represent each pixel. An image with depth 1 (monochrome image) uses one bit per pixel and allows only two colors (represented as 0 and 1 respectively), depth of 2 allows $2^2 = 4$ different colors encoded as 00, 01, 10, and 11, depth of 8 means that each pixel may have any one of $2^8 = 256$ colors, and so on. The greater the depth, the more subtle an image (if your computer can display them) and the more memory required to store the image. In ordinary situations, you don't need to be concerned about depth, and address all your messages to class Image.

The most common ways of creating an image are capturing an image on the screen or reading it from a file. We will see that cursors, which use an Image as a part of their definition, often create an image by specifying the individual pixels.

Example 1: Capturing an image and painting it in a window

Problem: Write a program to let the user capture an image from the screen and paint it in the currently active window. The upper left corner of the image should be located in the center of the window. Figure 12.4 shows what the outcome might look like.

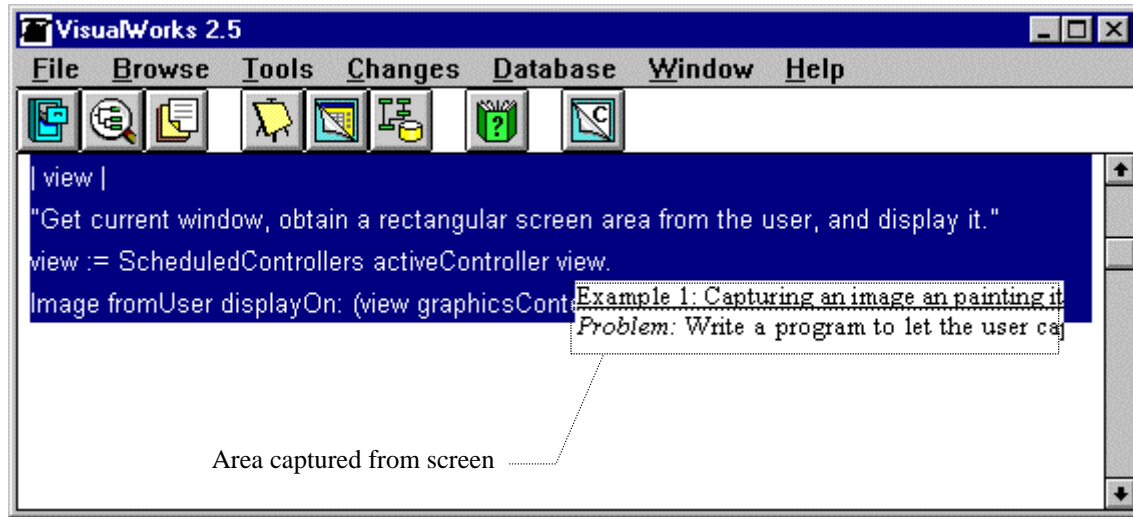


Figure 12.4. Example 2: Displaying image from user in the active window.

Solution: To get an image from the user send message `fromUser` to `Image`. To find the currently active window, ask `Window`. To find its center, ask for its bounds rectangle and its center. Finally, get its graphics context and display the image. The graphics context `displayOn:at:` message displays the image at the specified point. The whole solution is as follows:

```
| view |
"Get current window, obtain a rectangular screen area from the user, and display it."
view := Window currentWindow.
Image fromUser displayOn: (view graphicsContext) at: (view bounds center)
```

The following example is from the library,

```
ScheduledWindow new
    component: ((BorderDecorator on: Image fromUser)
        useHorizontalScrollBar);
    open
```

It opens a default-size window showing an image captured from the screen by the user. Note how we specify the image as the window's component. We can improve on this code a bit by making the window the same size as the captured window as follows:

```
| image |
ScheduledWindow new
    component: ((BorderDecorator on: (image := Image fromUser))
        useHorizontalScrollBar);
    openIn: ((10@10) corner: image bounds extent)
```

Example 2: Image from file

Problem: Write a code fragment to read a graphics file and display it in a scheduled window.

Solution: Abstract class `ImageReader` can read a limited number of image file formats, delegating its conversion to an `Image` to a subclass corresponding to the encoding. Sending message `fromFile:` to `ImageReader` returns an object containing the image; the image can then be extracted with message `image` and displayed in the usual way. As an example,

```
(ImageReader fromFile: 'C:\MSOffice\Clipart\flower.bmp') image
    displayOn: Window currentWindow graphicsContext
```

displays an image from the specified file. Message `display`: displays the image in the upper left corner (origin) of the window but `displayOn:at`: allows you to specify the position.

Cursor

A cursor is the visual representation of the current position of the mouse on the screen. On first thought, a cursor is just an image that can be moved around. On closer examination, cursors are partially transparent, showing some of the underlying background as the cursor moves across the screen. This is achieved by combining the cursor's image with a *mask*, a concept discussed in more detail later.

Class `Cursor` defines more than 20 predefined cursor shapes (such as `write`, `hand`, and `crossHair`) and some interesting cursor behaviors. In addition, the user can create new cursors as well.

As an example of a built-in cursor, the cross hair cursor is defined as a class variable in class `Cursor` and initialized as follows:

initCrossHair

```
CrossHairCursor := (self
  imageArray: #(
    2r0000000000000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0111111111111110
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000010000000
    2r0000000000000000
    2r0)
  maskArray: #(
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r1111111111111111
    2r1111111111111111
    2r1111111111111111
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0000000111000000
    2r0)
  hotSpot: 7@7
  name: 'crossHair')
```

The method defines the four `Cursor` instance variables: the image, the mask, the position of the hot spot, and the cursor name. The *hot spot* is the `Point` returned when a program requests cursor position and represents the mouse position on the screen. The individual pixels of the image and the mask are specified as binary numbers such as `2r0000000111000000` (2r means radix 2 - binary representation). The 1's in the *image* array are the black pixels and 0's are white pixels and when you half close your eyes and look at the

pattern of 0's and 1's, you can see the crosshair shape easily. The 1's in the *mask* define the pixels that are opaque (*not* transparent), in this case the crosshair and a one-pixel band around it. 0's are transparent and allow you to see the background behind the cursor. You can use this same method to define your own cursors and Appendix 4 shows how you can create a cursor with the Image Editor tool.

Class `Cursor` is typically accessed to display a special cursor during a special operation. As an example, VisualWorks displays a special cursor while reading a file, while performing garbage collection, or while waiting for the user to select a point on the screen (for this, it uses the `crossHair` cursor above). The two common messages for changing the cursor are `showWhile: aBlock` and `show`. The `showWhile:` message changes the cursor for the duration of execution of its block parameter, and changes it back to its original shape when `aBlock` executes. A typical example of its use is in the following method from class `FileBrowser`:

hardcopyStream: aStream

"Use the 'wait' cursor while printing aStream. "

```
Cursor wait
    showWhile: [aStream contents asText asParagraph hardcopy]
```

Method `show` also changes the cursor but the programmer must remember to change it back. Its typical use is shown in the definition of `showWhile:` itself:

showWhile: aBlock

"While evaluating aBlock, make the receiver be the cursor shape. Then revert to the original cursor."

```
| oldcursor |
oldcursor := self class currentCursor.
self show.
^aBlock valueNowOrOnUnwindDo:
    [oldcursor show]
```

Pixmap – drawing surfaces behind the scenes

A `Pixmap` is a display surface and you can 'display' on it, just as on the screen. However, displaying on pixmap does not show on the screen until you ask the screen's graphics context to display the pixmap. Pixmap are thus used to construct displayable rectangles before they are displayed on the screen. The typical procedure is to transfer an `Image` to a `Pixmap`, modify it by a mask to display only some selected part of it, and ask a graphics context to display the result. We will show an example of this shortly. Pixmap can also be created by copying the contents of the clipboard of your machine using the `clipBoard`

Extracting a part of an image - masks

To display a part of an image, transfer the original image to a `Pixmap`, combine it with a `Mask`, and display the result. If the desired mask is rectangular, a mask is not necessary because the `Pixmap` can do rectangular clipping itself. We will now describe and illustrate both techniques.

Displaying a rectangular part of an image without using a mask

1. Create the image as a `Pixmap` object or convert an `Image` to a `Pixmap` using `asRetainedMedium`.
2. Construct the clipping rectangle.
3. Apply the clipping rectangle to the `Pixmap` and extract the corresponding area as `Image`. Use message `completeContentsOfArea: clippingRectangle`.
4. Display the obtained `Image` on a graphics context.

Example 3: Extracting a rectangular portion of an image without using a mask

Problem: Display the image from the previous example and its lower half side by side (Figure 12.5).

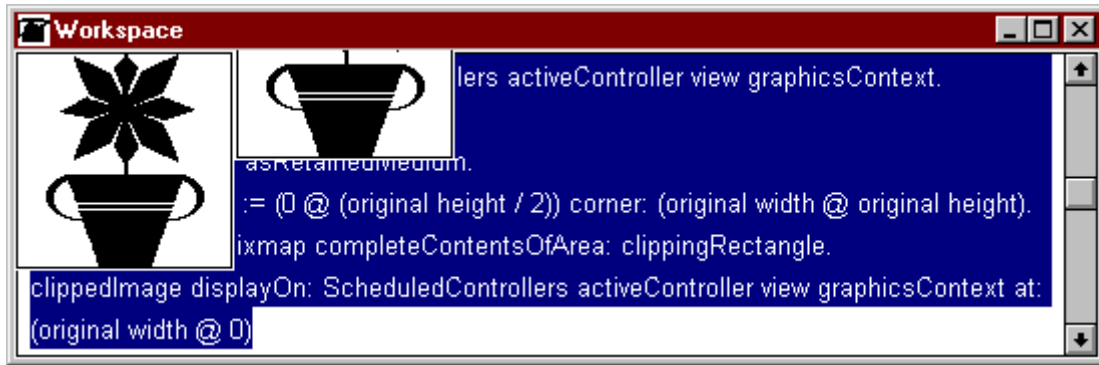


Figure 12.5. Example 3: Image from the Microsoft Office clip art library and its rectangular part.

Solution: Following the procedure described above and using a .bmp file, the solution is as follows:

```
| gc clippingRectangle clippedImage original pixmap |
"Get image from file."
original := (ImageReader fromFile: 'C:\MSOffice\Clipart\flower.bmp' ) image.
"Display original image on window's graphics context."
gc := Window currentWindow graphicsContext
original displayOn: gc.
"In memory, construct a Pixmap containing the original image."
pixmap := original asRetainedMedium.
"Construct clipping rectangle."
clippingRectangle := (0 @ (original height / 2)) corner: (original width @ original height).
"Extract area corresponding to rectangle from pixmap."
clippedImage := pixmap completeContentsOfArea: clippingRectangle.
"Display the clipped image on the graphics context next to the original."
clippedImage displayOn: gc
                    at: (original width @ 0)
```

An alternative approach is to use a Mask. This method is more powerful because it allows the clipping area to have any shape. During the discussion of Cursor, we saw that a mask acts as a decal – a display area painted in such a way that it is transparent in some places but opaque in others. A mask is thus like a surface covered with a 'paint' that is either transparent or opaque and this is why the paint is called *coverage value*.

Extracting a non-rectangular part of an image with a mask

1. Create the image as a Pixmap object or convert an Image to a Pixmap using `asRetainedMedium`.
2. Construct the clipping mask. The shape drawn on the mask will be the transparent area.
3. Display the Pixmap overlaid with the mask on the graphics context using `copyArea:from:sourceOffset:destinationOffset:`. You can control the position of the mask with respect to the pixmap by the argument of `sourceOffset:`, and the position of the pixmap with respect to the display area by the argument of `destinationOffset:`.

Example 4: Extracting a triangular portion of an image with a mask

Problem: Display an image from user and its above-the-diagonal part in the current window (Figure 12.6). Shift the clipped part so that its origin coincides with the corner of the original image.

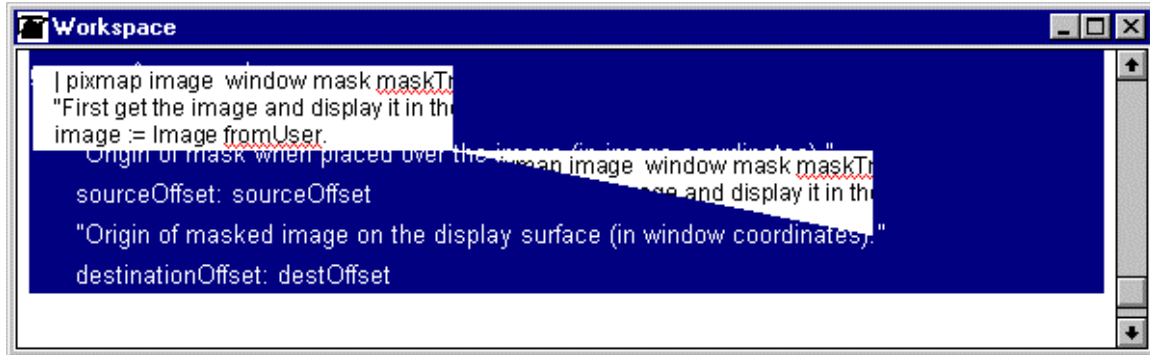


Figure 12.6. Example 4: Bitmap image clipped with a mask.

Solution: Following the steps listed above we create a Pixmap, a Mask containing the masking triangle, and display the combined image and mask on the graphics context of a window:

```
| pixmap image window mask maskTriangle gc imageRectangle destOffset sourceOffset |
"First get the image and display it in the current window."
image := Image fromUser.
window := Window currentWindow.
gc := window graphicsContext.
gc display: image at: 10@10.
"Now start constructing the masked image. First transfer image to pixmap."
pixmap := image asRetainedMedium.
"Find the bounding box of the image and use it to construct mask rectangle."
imageRectangle := image bounds.
mask := Mask extent: imageRectangle extent.
maskTriangle := Array with: 0 @ 0
                     with: (imageRectangle width) @ 0
                     with: imageRectangle width @ imageRectangle height.
"Display the triangle on the mask."
mask graphicsContext displayPolygon: maskTriangle.
"Display masked image on window's graphics context at an offset."
sourceOffset := 0@0. "No offset from pixmap origin to mask origin."
destOffset := imageRectangle extent + (10@10). "Window origin to pixmap origin offset."
gc copyArea: mask
  from: pixmap graphicsContext
  "Origin of mask when placed over the image (in image coordinates)."
  sourceOffset: sourceOffset
  "Origin of masked image on the display surface (in window coordinates)."
  destinationOffset: destOffset
```

Main lessons learned:

- An Image is a pixel-by-pixel representation of an image displayed on screen or printed by printer.
- An Image may use one of several depths, resulting in greater or smaller color variety and a corresponding increase or decrease of memory requirements.
- An Image can be read from a file, copied from the screen, or constructed from pixels.
- Class Cursor defines the visual representation and behaviors of the mouse cursor.
- Cursor shape is defined by a 16@16 image and a 16@16 mask that determines which part of the cursor is transparent.
- An Image can be clipped to any shape with a mask.
- A Pixmap is a display surface stored in memory but not necessarily displayed on the screen.

Exercises

1. You may have noticed that our image from use in Example 1 does not show the whole rectangle selected by the user. Why is it? Can you correct this shortcoming?
2. In Example 4, we set the offset of the origin of the mask at the origin of the pixmap. What happens when you use a non-zero offset?
3. Write a simple clip art browser that finds all .bmp files on a user-specified disk drive, lists their names in a multiple choice dialog, and displays user's selection in a new ScheduledWindow.
4. It would be nice if CoverageValue could handle any values between 0 and 1, making pixels less and less transparent as the value moves from 1 to 0. A reasonable approximation of this behavior is to make some of the pixels transparent and leave the remaining pixels opaque. As an example, if the coverage value of all pixels in a 10 by 10 rectangle is 0.6, we could make 60 percent of all pixels opaque and 40 percent transparent using random assignment of 0's and 1's in proportions corresponding to the specified coverage value. Implement this idea.
5. Explore class Pixmap, in particular its comment and displaying protocol. Write a summary.
6. Explore class Mask and write a summary.
7. Explore class Image, in particular its bit accessing and image processing protocols. Write a summary.
8. Write a method that creates a smooth visual transformation of one image to another. Allow several styles of transitions such as left to right, top to bottom, and so on.

12.5 Models, views, and controllers revisited

Although the graphics context, display surfaces, images, and the other concepts presented above provide a lot of power, they don't address three essential aspects of user interfaces: interactive user control, window damage repair, and automatic dependency on domain data. We will now explain what we mean by these concepts and how VisualWorks deals with them.

- By *user control*, we refer to the ability of the user to interact with the application using the mouse and the keyboard. Clearly, the programs that we wrote in the previous sections did not allow any user interaction with the painted objects and images. We will see shortly that the concept of a *controller* provides a mechanism for user control.
- *Automatic damage repair* is another GUI feature that we take for granted: If a window is 'damaged', for example collapsed to an icon and then expanded, or covered by another window and then uncovered, or simply resized, we expect that the damage will be automatically repaired. In other words, we expect a certain permanence of the GUI. You may have noticed that the drawing and painting in our example programs did not have this permanence. If you painted something on the sample window and then collapsed and restored it, the painting was gone. We will see that the basic mechanism for automatic repair can be obtained by subclassing views to the *View class*.
- Finally, we expect that if the value of the *domain model changes* in a way that should be reflected in the view, the user interface will automatically adjust to this change. As an example, if the window displays a

circle and the application changes its diameter, we expect that the circle will be automatically redrawn. The previous sections do not give any hints as to how this could happen but we will see that the principle of *dependency* implements this mechanism.

In the rest of this section, we will explain the principles of views, controllers, and their dependency on models. The remaining sections will then explain the basic details and demonstrate how they can be used to build new UI components by using the 'any component' widget of the UI painter called the *view holder* or the *subview*.

As we already know from Chapter 6, the principle of user interface components in Smalltalk is the *model-view-controller (MVC) paradigm*. According to this principle, every UI component that represents a value of a domain object uses this value as its *model*, the graphical representation of the model is the responsibility of a *view*, and the object that manages user interaction within the boundaries of a view is the view's *controller*. We will now outline how the MVC paradigm implements the three UI operations listed above.

User interaction

User interaction via mouse and keyboard is constantly monitored by the operating system. When an event occurs, the operating system sends information about it to the running application, in this case VisualWorks. After some opening message sends, an instance of a concrete subclass of Event sends a message corresponding to the event that occurred to the active controller, usually the controller responsible for the screen area under the cursor. The message contains information about the kind of event that has occurred and its parameters. If the controller is interested in the event, it must have its definition of the method, otherwise a 'do nothing' hook definition inherited from class Controller is executed.

As an example, the controller of a chess board view will need its own definition of `redButtonPressedEvent`: that will respond when the player presses the red button over a chess board square.

Damage repair

When a window or its part is damaged, it records the smallest rectangle containing the damage. When processing priority allows it, the window redraws the damaged rectangle or the union of all damage rectangles. To do this, the window constrains its GraphicsContext to the clipping rectangle corresponding to the damaged area and sends a message to its components, asking them to redraw themselves.

Unless the window consists of only one widget, it contains a 'composite part' containing other parts, as in Figure 12.7. Damage repair sends message `displayOn: graphicsContext` down the tree to all parts of the window and eventually reaches all views. Each view must have a definition of `displayOn:` (possibly inherited) and this definition redraws the view within the limits of the clipping area of the graphics context.

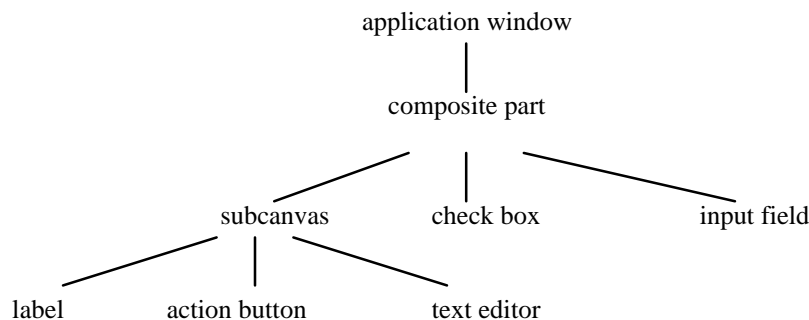


Figure 12.7. Simplified structure of a multi-component window.

Redrawing caused by a change in the model

In an active application, the value of a model frequently changes. As an example, a stopwatch application using integer seconds as its model changes the value of the model every second. There are two ways in which this change may be translated into a view change:

The preferred approach is to use *dependency*, already introduced earlier. To use dependency, the programmer includes a *changed* message in each method that changes the model in a way that affects the view. In the chess example (Figure 12.8), the model method that changes the configuration of pieces on the board when a player makes a move calculates new piece positions and sends *changed* to self¹. In response to this, the model automatically sends the *update* message to each of its dependents - in this case the chess board view. The chess board view's customized *update* method then asks the model for the necessary data and redraws itself appropriately. To redraw itself, the view usually sends *invalidate* to self and the *invalidate* message travels up the component tree to the window at the top, and the window then sends *display* down the component tree in the same way as after a request for damage repair.

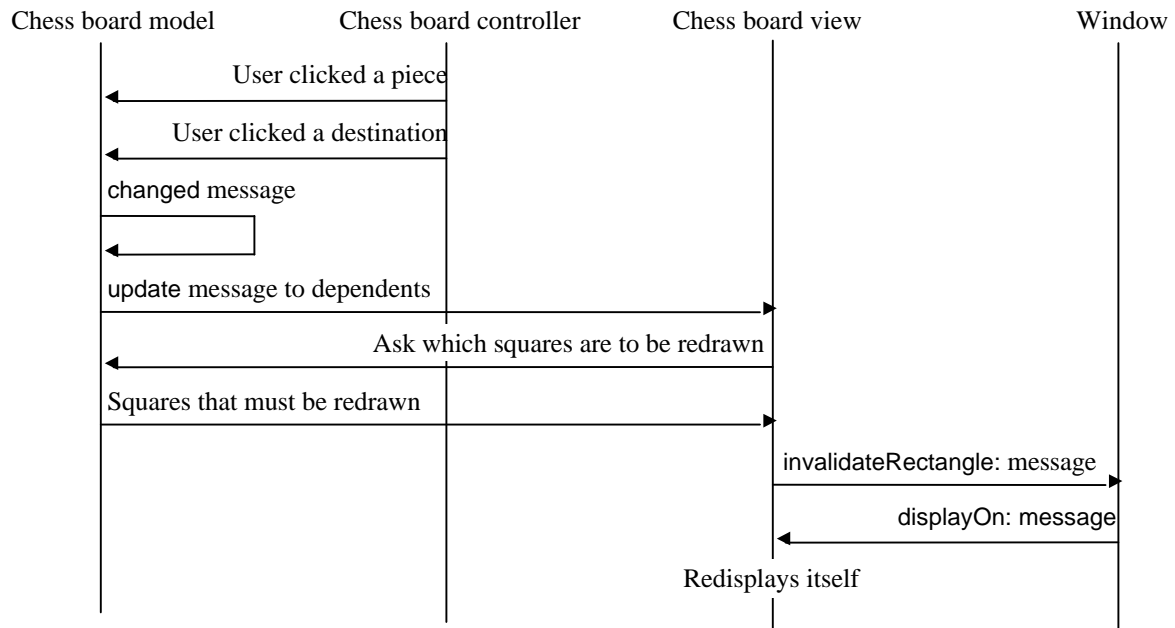


Figure 12.8. Communication involved in moving a chess piece from one square to another. Redrawing based on dependency and changed message is assumed.

The other way in which a model change may be translated into view redrawing is to send the *invalidate* message directly from the model's method when the change occurs. The result is the same as when we use dependency but the implementation is less centralized.

It is useful to note that invalidation provides two refinements. One is that we can specify whether the redrawing of the view must occur immediately or not. (This gives rise to the concept of 'lazy repair'.) The message may also specify that only a part of the view rather than the whole view should be redrawn and this can greatly improve the behavior of the user interface. As an example, when the position of a chess piece changes, only the start and the destination squares need to be redrawn. (The exceptions are capturing 'en passant' which affects three squares, and castling where four squares must be redrawn.) Invalidating only two squares instead of redrawing the whole board means that only 2/64 of the view must be redrawn and although this does not speed up the process, it eliminates unpleasant flashing that would otherwise occur. We use this technique in our implementation of the chess board in Appendix 3.

¹ The best way to do this is to change the aspect only via an accessing method that includes the *changed* message.

After this general introduction, we are now ready for details and examples. In the next two sections, we will explain how to create views using the view holder widget and how to create controllers. A larger example showing the implementation of the chessboard is given in Appendix 3.

Main lessons learned:

- The principle of Smalltalk's user interfaces is the separation of domain objects, display objects, and user interaction objects into a model, a view, and a controller. This principle is called the MVC paradigm.
- The model of a visual component holds the displayed domain object and treats the view as its dependent. When the model changes in a way that should affect the view, it sends itself a *changed* message which triggers notification of the view and any other dependents via *update* messages. The view is responsible for having an appropriate definition of *update* to redisplay itself.
- The view is responsible for drawing the model's visual representation on the screen. It should be a subclass of *View* to inherit basic damage repair mechanism and connections to its model and controller.
- The controller is responsible for handling mouse and keyboard events within the view's limits.

12.6 Creating UI components with the view holder widget

Older versions of Smalltalk did not have user interface painters and the only way to create a graphical user interfaces was programmatic. With the current technology, the UI Painter almost eliminates the need to program views and controllers because standard widgets are available from the Palette and non-standard UI components can be created rather easily with the *view holder (subview)* widget. (We will use the term view holder for the widget, and subview for the visual component displayed in the view holder.)

The process of creating a new component using the view holder widget is simple. It consists of painting the view holder on the canvas, defining its properties, creating the view that it will display, and defining its controller. We will now demonstrate the procedure on two simple applications that do not allow user interaction and therefore do not require a custom controller. In the next section, we will then give an example that requires a custom controller and show how to design it.

Example 1: Image display.

Problem: Design and implement an application with the user interface in Figure 12.9. Before the window opens, the program asks the user to select two images on the screen. When the window opens, the view shows help text and when the user clicks one of the image buttons, the view displays the corresponding image.

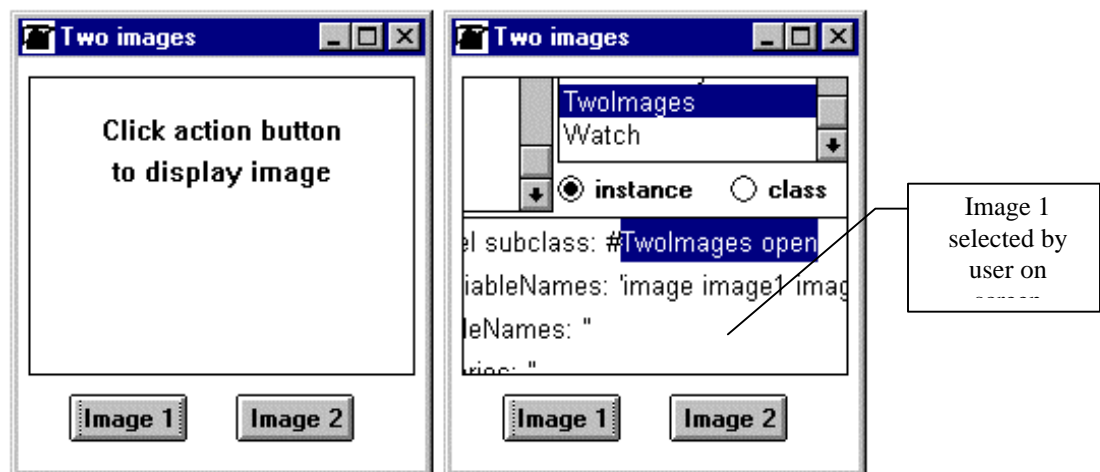


Figure 12.9. Example 1: Desired user interface when the application first opens (left), and when the user clicks an image button (right).

Solution: We will solve this problem by using dependency. The application model will be the model of the subview and when the user clicks an action button, the application will execute the `changed` message. This will trigger the `update` message in the subview which will then request the image from its model (the application) and invalidate itself. This will cause the subview to redraw itself.

The solution requires an application model (class `TwoImages`) and a view class (class `ImageView`) for the subview displayed by the view holder. Before discussing the details of implementation, we will first describe how to create the view holder:

Creating a view holder

1. Create the canvas and paint the view holder (Figure 12.10).
2. Define view holder properties, at a minimum the `View` property which is the name of the method that returns the subview to be displayed in the view holder. We called this method `imageView`.
3. Install the canvas.

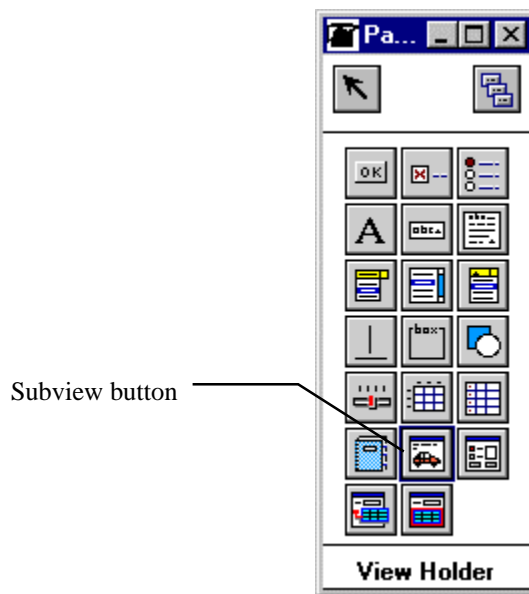


Figure 12.10. The view holder action button on the UI Palette.

Design

Class `TwoImages` is the application model class. It will have instance variables to hold the subview (`imageView`), the currently displayed image (`image`), and the two images selected by the user on the screen (`image1` and `image2`). During initialization, `TwoImages` will ask the user to select two rectangles on the screen and save the corresponding images in `image1` and `image2`. When the user clicks an action button, the corresponding action method assigns the appropriate image to `image` and sends the `changed` message.

Initialization must also create an instance of a view (class `ImageView`), assign it to `imageView`, and assign itself as its model. (ApplicationModel inherits dependency from its superclass `Model`.) The definition of `initialize` is thus as follows:

initialize

```
"Get two images from user and define subview."  
Dialog warn: 'Select the first image.'
```

```
image1 := Image fromUser.  
Dialog warn: 'Select the second image.'  
image2 := Image fromUser.  
imageView := ImageView new.  
imageView model: self    "Create dependency of the subview class on the model."
```

The action method for *Image 1* assigns image1 as the current image and triggers the change mechanism:

```
image1  
"Assign image1 as current image and send change notification to dependent (subview)."  
image := image1.  
self changed
```

and image2 is similar. The remaining methods in *TwoImages* are only accessing messages for image and imageView.

Class ImageView

Class *ImageView* implements the subview. As in all situations involving the view holder widget, the superclass of *ImageView* will be *View* because it implements the damage repair mechanism.

When a view is being built by the builder, a new instance is created and in the process, message *defaultControllerClass* is sent. Our view does not allow any user interaction and we will thus return the *NoController* class - a controller that ignores all input events:

```
defaultControllerClass  
^NoController
```

Note that a controller is required even if it is not active.

When a change occurs in the model (*TwoImages*), the model sends itself the *changed* message which then sends an *update* message to its view holder dependent. In response to this, the view holder must redraw itself, usually by sending itself the *invalidate* message which travels up the component tree of the window and causes the window to send a *display* message with its graphics context clipped to the damage area down the tree again. The definition of *update* is thus simply

```
update  
"Trigger a redisplay."  
self invalidate
```

but since this behavior is inherited, we don't need this method at all.

The only remaining method (and the method that does all the work) is *displayOn:* which redisplay the contents of the view holder whenever it is required (during initial display, during each model change, and during invalidation). To display itself, our subview must ask its model for the current image and if the image is not nil, ask the graphics context to display it. Since we have not initialized *image*, its value is initially nil. In this case, we will display a help text in the center of the subview. To get the proper location for the text, we will ask the subview for its bounding rectangle, ask the graphics context to measure the width of the text, and display the text at an appropriate place. The whole definition is as follows:

```
displayOn: aGraphicsContext  
"Display yourself on the graphics context supplied by the window."  
| string width center |  
model image isNil  
    ifTrue: "Display help text."  
        [center := self bounds center.  
        string := 'Click action button'.  
        width := aGraphicsContext widthOfString: string.  
        aGraphicsContext displayString at: (center x ( width/2)) @ 30).  
        string := 'to display an image.']
```

```
width := aGraphicsContext widthOfString: string.  
aGraphicsContext displayString at: (center x ( width/2)) @ 50)]  
ifFalse: "Display currently selected image."  
[aGraphicsContext displayImage: image at: 0@0]
```

The program is now fully functional. Note that unlike our previous programs it automatically repairs window damage due to collapsing, reshaping, and other window events.

Improvements

We know that it is better to centralize shared behavior and in our case both image methods share the changed message which is always executed when image changes. In such a case, image should be changed by an accessing message which sends the changed message. As a consequence, image1 should be

```
image1  
self image: image1
```

and the corresponding accessing method should be

```
image: anImage  
image := anImage.  
self changed
```

We conclude that the process of creating a subview and inserting it into a canvas is quite simple and can be summarized into the following three steps:

1. Paint the *canvas* with a view holder widget, specifying the name of the application model method that returns the corresponding view object as its *View* property.
2. In the *application model*,
 - a. create an instance variable to hold the view displayed in the view holder, and the corresponding accessing method specified as the *View* property in Step 1
 - b. initialize the view holder instance variable to an instance of the subview class and specify the application model (self) as the view's model (needed only if you use dependency)
 - c. when using dependency, every method that changes the model and affects the view must send changed to self. This is best implemented by busing accessing messages and including changed in them. When not using dependency, send self invalidate to the view when a change of the subview is desired. You can also redraw parts of the view selectively using invalidateRectangle: or invalidateRectangle:repairNow:.
3. In the definition of the subview class
 - a. specify View as the superclass
 - b. define a controllerClass method to return the class of the desired controller
 - c. define an update method if you use dependency to control the subview; if updating consists of sending invalidate and nothing else, this behavior is inherited and update: is not required.
 - d. define a displayOn: method (always); it generally obtains data from the model (the application) and uses it to redraw itself

As another illustration of the procedure, we will now do an example that shows how to display geometric shapes. The problem will give us an opportunity to see that geometric objects are not displayable in their raw form and require additional care.

Example 2: Displaying geometric figures

Problem: Implement an application with the user interface in Figure 12.11. When the user selects a radio button and clicks *Draw*, the window displays the corresponding geometric shape.

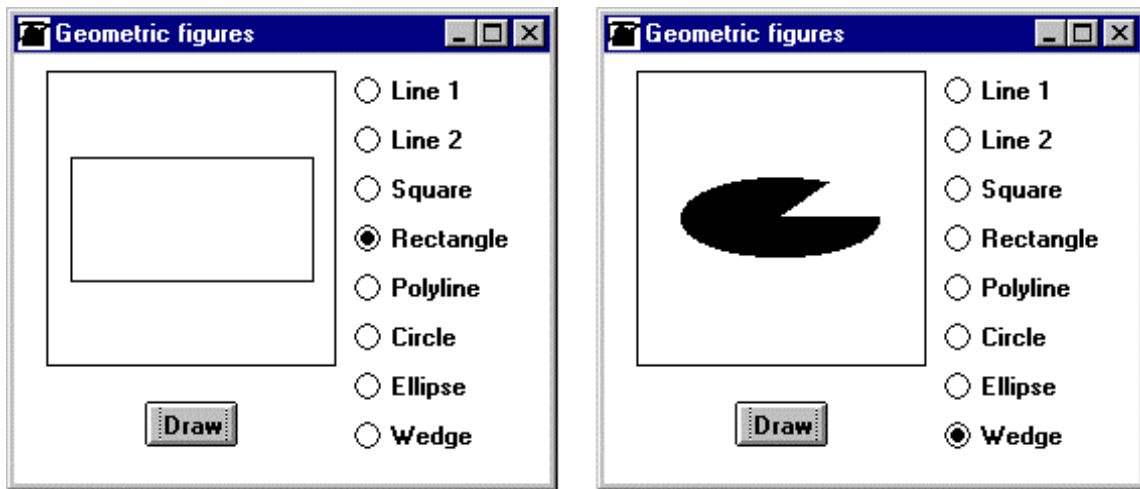


Figure 12.11. Example 2: Display of stroked rectangle (left), display of filled wedge (right).

Solution: We will use a subview with NoController to implement the drawing area. The application model (class GeometricFigures) keeps reference to the view in instance variable drawingView, and aspect variable object holds the current radio button selection. The draw action method will use dependency and send changed to the model. The corresponding update method will only invalidate the view so we don't have to define it. update will trigger displayOn: which will ask the application model for the selected geometric object (derived from the aspect value of the selected radio button), and display it. Implementation of the corresponding methods is as follows:

Initialization of GeometricFigures

initialize

```
"Select the top radio button, create a binding to the subview, and define yourself as the subview's model."  
object := #line1 asValue. "Initial selection when the window opens."  
drawingView := GeometricView new.  
drawingView model: self
```

The action method of the *Draw* button simply triggers the dependency-based update mechanism:

draw

```
"To redraw the subview, trigger the dependency mechanism."  
self changed
```

The display:at: method triggered by invalidate is defined in GeometricView as follows:

displayOn: aGraphicsContext

```
"Obtain currently selected geometric object and display it in the center of the view."  
aGraphicsContext display: model currentObject at: self bounds center
```

The general-purpose display:at: method in GraphicsContext can display any displayable object. In our earlier example, we used shape-specific messages such as displayLineFrom:to: but this strategy would be very awkward in this case. The currentObject message is defined in GeometricFigures and uses the radio button selection to get the desired object as follows:

currentObject

```
"Use currently selected radio button value to obtain the corresponding geometric object."  
^self perform: object value
```

If the selected button is, for example, *Rectangle* (aspect object == #rectangle), the last line executes

self rectangle

and to complete our implementation, we must define the methods that calculate and return the appropriate displayable geometric objects. A very important feature of all of them is that `display:at:` cannot display geometric objects in their raw state and the object must first be ‘wrapped’ by conversion messages `asStroker` (for empty figures such as the rectangle on the left of Figure 12.11) or `asFiller` (for filled figures such as the wedge on the right of Figure 12.11). As an example, the method creating the wrapped rectangle is

rectangle

“Return a displayable rectangular border. The coordinates are calculated on the premise that the view displays the object in the center.”

```
^(Rectangle origin: -60 @ -30 corner: 60 @ 30) asStroker
```

and the method creating the wrapped wedge is

wedge

“Return a displayable rectangle. The coordinates are calculated on the premise fact that the view displays the object in the center.”

```
^(EllipticalArc  
  boundingBox: (-50 @ -20 corner: 50 @ 20)  
  startAngle: 0.0  
  sweepAngle: 300.0) asFiller
```

We leave the remaining methods as an exercise.

Main lessons learned:

- Custom components can be created with the view holder widget which contains a subview with a controller.
- The only required property of a view holder is *View*. Its value is the name of the method that returns the subview’s view object.
- In order to inherit the damage repair mechanism, the subview should be a subclass of *View*.
- The view class must define the `displayOn:` message which is sent automatically whenever the window needs to draw or redraw the subview. Also required is message `defaultControllerClass`.
- If response to changes of the view’s model uses dependency, the view must also define an `update` method. If the only required action is `invalidate`, the behavior is inherited and `update:` is not needed.

Exercises

1. Does *View* define a default controller class? If so, what consequences does it have for view design?
2. Describe in detail what the `asStroker` and `asFiller` messages do. Inspect the result of sending such a message to a rectangle.
3. Write a description of wrappers.
4. Reimplement Example 1 using direct invalidation instead of dependency.
5. Extend Example 2 by adding a group of radio buttons labeled *stroked* and *filled* that allow the user to select how the geometric object should be displayed.
6. Which class defines the `update` message inherited by views?
7. Record a complete trace of an update of a view.

12.7 Controllers

Controllers are responsible for managing user input and the class library contains many controller classes for various kinds of widgets and tools. Only a few, however, are of interest to most programmers. The essence of the controller class hierarchy (with controllers commonly needed in applications shown in bold) is as follows:

```
Object ()
  Controller ('model' 'view' 'sensor')
    ControllerWithMenu
      ModalController
      ParagraphEditor
        TextEditorController
      SequenceController
    LauncherController
    MenuItemController
    NoController
    ScrollBarController
    StandardSystemController
```

The Controller class on the top defines all essential instance variables and behaviors and is often used as the direct superclass of custom controllers. Its instance variables include *model* and *view* which are responsible for communication with the other two components of the MVC triad, and *sensor* - an instance of a concrete subclass of the abstract class *InputSensor* which handles mouse and keyboard input and can provide information on such parameters as cursor coordinates.

In the past, all VisualWorks controllers used *polling*, a technique which constantly tests whether the user performed an input event, and takes the appropriate action if necessary. Polling requires extra processing time and this may cause some of several input events issued in quick succession to be missed. The preferred modern implementation of controllers is thus *event-driven* where VisualWorks is immediately notified by the operating system when an input event takes place, and sends this information to a controller in the form of an appropriate predefined notification message². These notification messages are defined in the events protocol in class *Controller* and include *entryEvent:* and *exitEvent:* (automatically sent to the controller when the cursor enters or leaves the area of the controller's view), *doubleClickEvent:*, *redButtonPressedEvent:*, *keyPressedEvent:*, and many others.

All event messages have a single argument called *event* - an instance of an appropriate *Event* subclass such as *CloseEvent*, *KeyPressedEvent*, and *MouseMovedEvent*. The *Event* object contains information relevant for the event that triggers it such as the x and y coordinates of the current position of the hot spot of the cursor or the activated key. All of these predefined messages are *stumps* containing only an empty body, and concrete subclasses redefine them to perform whatever actions are necessary when the corresponding event occurs. In other words, event messages are hooks. We will see examples of them shortly.

Besides Controller, the most important controller classes are

- *StandardSystemController* - in charge of window interaction, in particular its <window> pop up menu
- *ControllerWithMenu* - has a *menuHolder* for the <operate> menu, and a *performer* (the object that performs menu commands)
- *TextEditorController* - used by the text editor widget
- *ParagraphEditor*- superclass of *TextEditorController* defining most of its functionality
- *NoController* - controller that does not respond to any user input

In this chapter, we are interested in controllers in the context of creating new UI components. We have already seen that this is done by using the view holder widget and defining a view-controller pair for it. In the previous section, we used *NoController* because our subviews did not allow any user interaction. In

² The actual implementation of events in VisualWorks is still based on polling but the outward appearance is event-driven.

the following example, we will create a UI component with an active user interface and show how to create an active controller that responds to user input.

Example: Subview with clickable hot spots

Problem: Implement an application that displays a bordered subview (Figure 12.12) equipped with an <operate> menu and capable of responding to mouse clicks. When the subview is empty, the menu contains only the *Add hot spot* command. When the view contains at least one hot spot, the menu also displays a *Remove hot spot* command.

When the user selects *Add hot spot*, the program requests a string (hot spot name) and the cursor changes to cross hair. (The cursor changes back to the original shape whenever it leaves the subview area and changes back to cross hair when it re-enters the subview.) Clicking the left mouse button inside the subview creates a 'hot spot' and displays it as a small red rectangle. The cursor then changes back to its original shape.

When the user clicks *Remove hot spot*, the cursor changes to cross hair and clicking the left mouse button over a hot spot displays a warning with the name of the hot spot and deletes the hot spot from the subview. (During this procedure, the cursor again changes to the original shape whenever it leaves the subview area.) After redisplaying the view, the cursor changes back to its original shape.

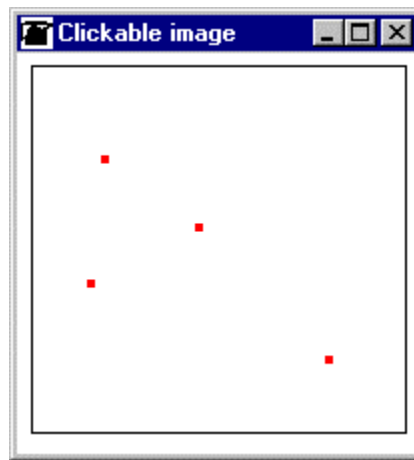


Figure 12.12. Example: Clickable subview with four hot spots.

Solution: We will need three classes - the application model (ClickableImage), the subview class (ClickableView), and its controller (ClickableController).

Class ClickableController

ClickableController will be a subclass of ControllerWithMenu because the specification requires an <operate> menu. It will define menu building methods, and event behaviors for subview entry and exit and for red button press.

To implement the desired cursor behavior, ClickableController must have an instance variable to hold the shape of the cursor before it changed to cross hair so that we can restore it upon exit from the subview or when a hot spot related operation ends. We will call this variable cursor. Since cursor is undefined before the first user action, the exitEvent: method will access it via the following lazy accessing method:

cursor

"Lazy initialization of cursor."

^cursor isNil

ifTrue: [Cursor current cursor]

"Get current cursor shape from the system."

```
ifFalse: [cursor]
```

The `enterEvent:` message is sent when the cursor enters the subview. It must ask the application model which command is currently executing (we will call this method `state`), and display the cross hair cursor if the command is not nil:

enterEvent: event

"If the model is executing the Add or the Remove command, change cursor to cross hair."

model state isNil

```
ifFalse: [cursor := Cursor currentCursor.  
Cursor crossHair show]
```

Since this method is defined in the subview's controller, it will only be executed when the controller is in charge and this happens only when the cursor is in the subview's area. The `exitEvent:` method resets the cursor if `state` is not nil. Note that we would get the same result if we did not check `state` at all:

exitEvent: event

"If the model is executing Add or Remove, reset cursor on exit from subview."

model state isNil

```
ifFalse: [cursor show]
```

The `state` method in `ClickableImage` returns `#addHotSpot`, `#removeHotSpot`, or nil, depending on the command currently being executed.

The handler of the red button click event (again executed only if the cursor is within the subview) asks the model to execute the operation corresponding to the current state (add/remove hot spot or do nothing), sending the coordinates of the cursor as the argument:

redButtonPressedEvent: event

"Send message with cursor coordinates to model."

|position|

```
position := self sensor cursorPointFor: event. "Extract position from event."  
model executeOperationWith: position           "Result depends on the state of the model."
```

The next task of the controller is to implement the menus. Since the menu depends on the state of the application, we will leave it to the application to construct the menu as necessary. We are thus finished with the controller!

Class ClickableView

Our next class is `ClickableView`, naturally a subclass of `View`. From the previous section, we know that its responsibilities include accessing its controller class, responding to `update` (we will again use dependency), and responding to `displayOn:`. The controller is `ClickableController`, hence

defaultControllerClass

```
^ClickableController
```

There is no need for `update` since it only invalidates and this is an inherited behavior. And the `display` message displays all hot spots held by the model as little red squares:

displayOn: aGraphicsContext

"Display all hot spots as red squares centered at hot spot coordinates."

aGraphicsContext paint: ColorValue red.

model hotSpots do: [:hotSpot|

```
aGraphicsContext displayRectangle: (hotSpot at: 1) -2 extent: 4 @ ]
```

Class ClickableImage

The last undefined class is ClickableImage - the application model. Its instance variables will be clickableView (for accessing the view displayed in the subview - the view holder's *View* property), hotSpots (a collection of two-element arrays containing hot spot coordinates and names), and state. We will also find useful two extra variables: hotSpotName will hold the name of a new hot spot, and controller will provide direct access to the controller for menu control - although the controller could be accessed via imageView, the need is frequent and 'caching' the controller in a variable is better.

Variable state requires closer attention. As we noted above, *Add* and *Remove* actions are triggered by message executeOperationWith: from the controller. The easy way to execute a message that depends on aSymbol is perform: aSymbol. Since the operation requires cursor coordinates, the perform message must be a keyword message with cursor coordinates. To satisfy the different possible states, we will use the following three values of state: nil (not executing a menu command), #addHotSpotAt:, and #removeHotSpotAt:, and define methods corresponding to the last two values.

We are now ready to implement ClickableImage. Initialization involves setting up instance variables and telling the controller that the performer of the menu messages (the executor of menu commands) will be the model. We must also initialize the menu:

initialize

```
hotSpots := OrderedCollection new.  
imageView := ClickableView new.  
imageView model: self.  
controller := imageView controller. "Construct and assign the controller."  
controller performer: self. "The performer of menu commands is the application model."  
controller menuHolder value: self menuWithAdd "The initial value of <operate> menu."
```

The following two methods construct the <operate> menu when there are no hot spots and the menu for a view with at least one hot spot. The definitions are as follows:

menuWithAdd

```
"Menu when no hot spots exist."  
| menuHolder |  
    menuHolder := MenuHolder new.  
    menuHolder add: 'add hot spot' -> #addHotSpot.  
    ^menuHolder menu
```

and

menuWithRemove

```
"Menu when at least one hot spot exists."  
| menuHolder |  
    menuHolder := MenuHolder new.  
    menuHolder add: 'add hot spot' -> #addHotSpot.  
    menuHolder add: 'remove hot spot' -> #removeHotSpot.  
    ^menuHolder menu
```

When the user clicks *add hot spot* or *delete hot spot* in, the controller sends addHotSpot or removeHotSpot to its menu performer - the ClickableImage object and our next step is to define these two methods. Method addHotSpot asks the user for the name of the new hot spot and changes state to #addHotSpotAt:. Control then passes into the hands of the controller (defined above) which tracks cursor entry and exit into the subview and sends executeOperationWith: to ClickableImage when the user clicks the red mouse button in the subview. This analysis leads to the following definition:

addHotSpot

```
"Start 'add hot spot' operation - get hot spot name and change state; leave the rest to the controller."  
    hotSpotName := Dialog request: 'Enter hot spot name.'  
    state := #addHotSpot "Name of method to be executed on click button event."
```

Method `removeHotSpot` only changes the state:

removeHotSpot

```
"Start remove hot ' operation - change state; leave the rest to the controller."  
state := #removeHotSpot "Name of method to be executed on click button event."
```

After the execution of any of these two methods, the application waits for the user to move the cursor or click the mouse button which then sends the appropriate mouse event to the controller. This triggers `redButtonPressedEvent:` defined above, which sends `executeOperationWith: aPoint`. As we already explained, the method simply performs the state symbol with a `Point` argument provided by the controller:

executeOperationWith: aPoint

```
"Execute state operation with point supplied by controller. Ignore if not in 'add' or 'remove' state."  
state isNil "If nil, we are not in executing a menu command – no action."  
ifFalse: [self perform: state with: aPoint.  
self changed]
```

The real work of adding and removing hot spots is performed by `addHotSpotAt:` and `RemoveHotSpotAt:` and by the dependency mechanism invoked by `changed`. Method `addHotSpotAt:` resets the cursor to what it was before it changed to cross hair. It then adds the new hot spot to the collection, makes sure that the `<operate>` menu now includes *remove* because we now have at least one hot spot, and resets state to nil (end of add state):

addHotSpotAt: aPoint

```
"Add hot spot to collection, reset state and cursor, update menu."  
controller cursor show.  
hotSpots add: (Array with: aPoint with: hotSpotName).  
controller menuHolder value: self menuWithRemove.  
state := nil
```

Method `removeHotSpotAt:` is slightly more complicated because it must check whether the clicked point lies within one of the red squares:

removeHotSpotAt: aPoint

```
"Check if aPoint corresponds to a hot spot and if it does, inform user, display name, remove hot spot from  
collection, reset state and cursor, and update menu if necessary."  
| hotSpot |  
controller cursor show.  
(hotSpot := self hotSpotAt: aPoint) isNil "Return hot spot under cursor or nil."  
ifTrue: [^Dialog warn: 'Not a hot spot'.  
"Display name of hot spot being removed."  
Dialog warn: 'Hot spot to be removed: ', (hotSpot at: 2).  
hotSpots remove: hotSpot.  
hotSpots isEmpty "Change menu if there are no more hot spots."  
ifTrue: [controller menuHolder value: self menuWithAdd.  
state := nil
```

The only remaining method is `hotSpotAt: aPoint` which checks whether `aPoint` corresponds to a hot spot. It returns a hot spot corresponding to the position if the cursor is in a small area surrounding the hot spot, and nil otherwise:

hotSpotAt: aPoint

```
"Is aPoint within 2 pixels from a hot spot center? If so, return the hot spot."  
^hotSpots detect: [:hotSpot | (aPoint - (hotSpot at: 1)) abs <= (2 @ 2)]  
ifNil: [nil]
```

Note the use of point arithmetic and comparison.

Main lessons learned:

- Although VisualWorks provides polling as one of the ways to program controllers, event driven controllers are now preferred.
- Every view must have a controller, even if it does not allow user interaction.
- Custom controllers are usually subclasses of Controller or ControllerWithMenu.
- Class Controller defines many stumpy event methods that are triggered by the operating system when an input event occurs. When defining a custom controller, redefine those event methods that must be handled by the subview.
- All event methods have an event argument which contains all necessary information about the event that has occurred.
- Class ControllerWithMenu adds a menu holder for the <operate> menu, and a reference to the performer of the menu commands. The performer is often the application model but the default is the controller itself.
- Class ParagraphEditor and its subclass TextEditorController add text editing and code execution menu and its support.

Exercises

1. Reimplement our Example without lazy accessing and evaluate both approaches.
2. Reimplement the hot spot example by replacing the array representation of hot spots with a class called HotSpot and consisting of a name and a rectangle with the appropriate dimensions.
3. Draw a scenario diagram showing what happens when the user clicks *add* or *remove* in our example.
4. Explore classes Controller, ControllerWithMenu, StandardSystemController, and NoController and write a description of their essential behaviors.
5. It is not always necessary to specify the performer of a menu because performer has a default. What is this default and what does this definition imply?
6. Explore classes ParagraphEditor and TextEditorController and write a description of their essential behaviors.
7. Explore event methods and event classes and write a description of their essential behaviors.
8. Method displayOn: in our example redraws the whole subview which is not necessary. If we just added a new hot spot or deleted an existing one, we only need to redraw the corresponding small part of the subview and this can be achieved by using invalidateRectangle: instead of invalidate. Modify the program to use this approach and comment on the result.
9. Add command *change background color* to the <window> menu. The command will display a multiple choice dialog listing all predefined colors and when the user selects one, it will change the color of the currently active window. (Notes: You must reinitialize class StandardSystemController after changing the menu - why? After changing the color, send display to the window for the change to take effect.)
10. The concept of a clickable view has uses beyond our simple example. List several applications that could use mouse sensitive areas and design a general purpose ClickableController suitable for as many uses as possible.
11. Develop an application for drawing curves defined by mathematical formulas. We leave the detailed specification to you but the general principles should be as follows: The user interface allows the user to enter a function as a one-argument block, the start and end points of the argument, and the color in which the curve is drawn. Several curves may be drawn in the window at the same time and on the same scale. The window is self-repairing.
12. The previous exercise suggests the idea of a curve-drawing widget, along the lines of the Calendar widget described in Appendix 2. Write the specification and develop such a widget. Test your widget by reimplementing the previous exercise.
13. Reimplement the Calendar widget using a subview.
14. The TextCollector class which is the model of the Transcript is a very interesting counterpart of the Text Editor because it maintains focus at the end of its text contents: All output to it is automatically

appended at the end. This is very useful in applications such as system logs or logs of network activity which must be maintained in the order in which they are generated. Unlike the Text Editor, however the TextCollector does not have its own button on the UI Palette and if you wish to use it, you must use other means. Give a detailed description of how you could incorporate Transcript-like behavior in an application window.

Conclusion

VisualWorks user interfaces are based on three kinds of objects: display surfaces, the objects being displayed, and the object doing the drawing and holding the drawing parameters.

Display surfaces can be divided into those dedicated to displaying on the screen and those dedicated to printing. Screen related display surfaces include windows, masks, and pixmaps. Among the several kinds of windows, ApplicationWindow is most important because it implements canvases drawn with the UI Painter. Pixmaps are used to construct displayable objects in memory before they are displayed on the screen, and masks are used to control which part of an underlying image is displayed and which is hidden or transparent.

Programming of a window is usually restricted to controlling window parameters such as window label or background paint. The programming of a pixmap usually consists of displaying an image on it and masking it with a mask. Sometimes, a pixmap is used just to construct an image, for example as a composition of geometric objects. The programming of a mask usually consists of displaying a shape object that defines the mask's transparent area.

The drawing of visual objects is performed by a GraphicsContext. This crucial class defines methods for displaying objects such as images, text, and geometric objects, and holds display parameters such as line width, text font, color, and clipping rectangle which defines the part of the window or subview used for display. All of these parameters can be modified programmatically at run time. GraphicsContext is essential for the design of custom GUIs because it implements all display.

Visual objects that can be drawn by a GraphicsContext on a display surface include strings, geometric objects, and images. Geometric objects can be drawn either as anonymous objects using display:at: or by object-specific messages such as displayRectangle:at:. In the first case, the object must first be wrapped as a stroker or as a filler. Object-specific methods such as displayRectangle:at: include this operation.

Although most applications of paint use instances of ColorValue and paint the surface or display the object in a single uniform color, paint can also be an instance of a Pattern. A Pattern is essentially an image used as a tile to wallpaper a rectangular area on a display surface.

Images are pixel-by-pixel representations of rectangular areas. Their public protocol is defined in the abstract class Image which transparently delegates work to its concrete subclasses that implement images for specific image depth. (Image depth is the number of bits used to represent the color of one pixel.)

Another object useful in GUIs is the cursor representing the position of the mouse on the screen. Class Cursor contains a variety of predefined cursor shapes, and new ones can be created by specifying cursor image, mask (defines the see-through area), hot spot (the Point returned when the cursor is queried for its position), and name. A convenient way to create a cursor shape is by using the Image Editor tool which contains a special command for saving an image as a mask.

To create a new GUI component such as a chess board or the face of a clock, select the subview widget (view holder) from the UI Palette, paste it on the canvas, and define at least its View property - the method that returns the view object displayed in the subview. As the next step, construct your view class, defining at least method displayOn: aGraphicsContext which is the basis of the display of the subview. The method displays on a GraphicsContext supplied by the application window, with its default parameters which can be changed in displayOn:. If you use dependency, you must also sometimes define an update method. In most cases, update sends invalidate to self and this behavior is defined in a superclass of View and inherited. If your update is more complex, for example requiring invalidateRectangle: you must define update in your view class. Invalidate messages travel up the component tree to the window which then calculates the clipping area of its GraphicsContext and sends displayOn: aGraphicsContext down the tree to

its components with the clipped `GraphicsContext`. Each component then redraws that part of itself which falls within the clipping rectangle, using the supplied default graphics context parameters or redefining them temporarily.

Older versions of VisualWorks used polling controllers which repeatedly queried the sensor for UI events such as mouse movement or button clicks. Since polling requires extra overhead and since UI events occur fast, this approach often missed UI events. Modern implementations still provide polling controllers but add event-driven controllers which obtain event notification directly from the operating system, eliminate the polling overhead, and thus provide better performance. All you have to do to create a custom event-driven controller is create a new controller class subclassed to `ControllerWithMenu` (if you require an <operate> menu) or `Controller` (if you don't need an <operate> menu) and redefine those of its event tracking methods that are important for your subview. If your subview does not have an active user interface, specify `NoController` as the default controller class.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

ApplicationWindow, **Controller**, **ControllerWithMenu**, *CoverageValue*, *Cursor*, **GraphicsContext**, *Mask*, **NoController**, *Paint*, *Pattern*, *Pixmap*, **TextCollector**.

Terms introduced in this chapter

controller - part of GUI responsible for user interaction via mouse and keyboard events

clipping rectangle - rectangle restricting display area

cursor hot spot - Point reported by cursor when it is queried about its position

damage repair - repainting of surfaces damaged by closing or overlaying a window

display surface - screen or printer page on which painting is taking place; memory representation in the case of mask

event-driven controller - controller whose UI event messages are triggered by the operating system (see also *polling controller*)

filler - filled geometric object (see also *stroker*)

graphics context - object that holds display parameters and performs the display

mask - shape overlaid over display area; determines which part of area is visible or transparent

image depth - number of bits used to represent one pixel

invalidation - the process of notifying a window that a view or its part should be redisplayed; the propagation of this request to the components

pattern - a form of paint, a tile that can be used to wallpaper a rectangular area

paint - a color used to paint a surface or a pattern used to tile it

pixel - picture element, an addressable dot on the screen

polling controller - a controller that repeatedly asks its event sensor for UI events and responds appropriately; outdated (see *event-driven controller*)

stroker - contoured geometric object (see also *filler*)

subview - contents of a *view holder*

view - object responsible for displaying a predefined or custom GUI component

view holder - widget used to create custom visual components

Chapter 13 - Processes and their coordination, additional UI topics

Overview

Up to now, we were developing applications that ran until completion, run for a while, switch to another application, run the other application, return to the original application, and so on. However, there are many applications that should run while other applications run, automatically taking turns with them, using their share of CPU time when they need it and returning control to other applications when they don't need it any more. Some examples of this are network communication (the network communication program must be ready to run at any time and give up control when it does not need the CPU any more), printing while another program is running, and getting data from the web while the user is doing something else. Simpler examples include a computer clock that displays time without interfering with other programs and losing seconds, a stopwatch running in parallel with other programs, and alarms that run in the background, marking time and producing screen notification at a time previously specified by the user.

There are even applications that require that several of their own components run 'simultaneously'. As an example, it is not unreasonable to design a simulation of an airport with multiple airplanes landing, taking-off and circling the airport in such a way that each airplane behaves as if it had its own computer supporting it. And the Smalltalk environments itself has several 'applications' running at the same time - if we consider, for example, the independent process of garbage collection as an 'application'.

All of these examples use the principle that a CPU does not have to be dedicated to a single program or a single thread within one program, but can switch between any number of programs or threads of execution in an organized way, giving the appearance that they all run independently of one another and in parallel. In computer science terminology, these independent threads of execution are called processes and this chapter gives an introduction of their implementation in VisualWorks Smalltalk.

Please note that this chapter is not a complete treatment of processes but rather an introduction that illustrates the concept on several intuitive problems.. We also use our examples to introduce some new aspects of the construction and control of custom UI components.

13.1 A stopwatch and the concept of a Process

Some very natural programming problems require splitting the use of the CPU among several threads of tasks. The following example illustrates one such a situation.

Example: Analog and digital stopwatch

Problem: Design a stopwatch with analog and a digital components as in Figure 13.1. Clicking *Start* starts the stopwatch, updating both the analog and the digital part in one-second increments. Clicking *Stop* freezes the display, and clicking *Reset* resets the display to 0 seconds. The stopwatch must have minimal effect on the operation of the rest of the environment in that it should be possible to run other applications while the stopwatch is running, and running another application should not stall the stopwatch.

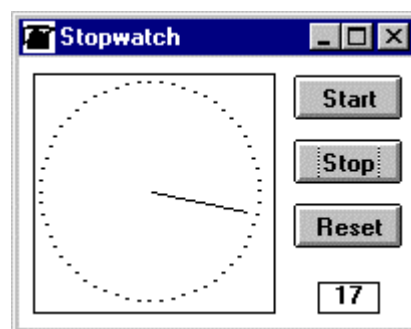


Figure 13.1. Stopwatch example.

Solution: The principle is straightforward: When the user clicks *Start*, the program begins an infinite loop that increments time every second and updates the display. *Stop* and *Reset* have obvious effects. The solution will require only an application model class (*StopWatch*) and a view class for the display of the analog watch (*WatchView*). Since the user cannot directly interact with the analog display, the subview does not need an active controller.

Although the principle is simple, the new problem is that the program must act independently of other programs.

To start, we painted the canvas and assigned the following properties: time is the Aspect of the input field showing digital time, watch is the Aspect of the view holder, and start and reset are the names of the action methods. We then proceeded to design and implement the two classes as described next.

Class WatchView

As a view class, *WatchView* requires a *displayOn:* method to display the analog face and handle damage repair. Updating will use dependency and method update:.

Method *displayOn:* will draw the seconds marks around the face of the clock, and the hand. The hand (but not the marks) must also be redisplayed when time changes and when the user clicks *Reset*. Displaying the hand in a new position requires erasing the current hand, and drawing a new one in the new position. Erasing can be done by redrawing the hand in the color of the background. We will implement this mechanism by four methods: two low-level methods *erase:* and *draw:* that draw and erase the hand, and two high-level methods *move:* and *reset:* that 'move' the hand to a specified second or to the starting position respectively. All four will take integer time in seconds as its arguments, and *move:* and *reset:* will combine erasing the hand in its current position and redrawing it in the new position. Method *move:* will be used during regular stopwatch operation, *reset:* for resetting.

Let's analyze the details of drawing. The marks around the face are drawn in one-second intervals. Since there are 60 seconds in a minute and one minute corresponds to a full circle on the stopwatch, mark locations are separated by $2\pi/60 = \pi/30$ radian increments. The point at which the mark is drawn is best obtained by using polar coordinates - the radius (constant) and the angle (60 increments of $\pi/30$ each). The angle of the hand is calculated similarly - by multiplying integer seconds by $\pi/30$.

We must now consider how to implement this operation efficiently because we want the display to take as little time as possible, partly to eliminate flashing, and partly to take away as little CPU time as possible from other running programs. Calculating π divided by 30 every time we want to display a mark or the hand is obviously unnecessary since we can pre-calculate this constant and cache it in an instance variable. Another constant can also be pre-calculated: The starting point of the stopwatch (0 seconds) is at the top of the circle, in other words, at - 90 degrees with respect to the x axis. 90 degrees corresponds to one quarter of a circle, in other words $2\pi/4 = \pi/2$. We will calculate this value during initialization and save it in another instance variable. We also need a variable to hold the coordinates of the center, the length of the hand, and the radius of the circle of the marks around the face. These variables must all be initialized before we can execute the *displayOn:* method.

Considering that the center of the subview can only be calculated when the subview has been created (either before or after the window opens), we will trigger this initialization from the *postBuildWith:* method in the *StopWatch* application model; it would not work if triggered from *initialize* which is executed before the subview is built. We will call this initialization method *setParameters* and its definition in the subview class *WatchView* is as follows:

setParameters

"Precalculate essential parameters of stopwatch geometry."

| bounds |

bounds := self bounds.

"Obtain square containing the subview."

center := bounds center.

radius1 := bounds width - 20/2.

"End point of hand."

radius2 := radius1 + 6.

"Circle on which marks are drawn around the face."

const2 := Float pi / 2.

"90 degrees in radians – corresponds to 15 seconds."

```
const1 := const2 / 15           "Separation of seconds in radians."
```

Method `displayOn:` which redraws the whole face of the digital clock with the marks and the hand is as follows:

displayOn: aGraphicsContext

```
"Display markings and hand."  
| theta |  
    "Draw marks around the face."  
    theta := 0. "Angle at which the next mark will be drawn."  
    1 to: 60 do: [:sec | endPoint |  
        theta := theta + const1. "Increment by  $\pi/30$ ."  
        endPoint := center + (Point r: radius1 theta: theta).  
        aGraphicsContext displayDotOfDiameter: 3 "Seconds mark."  
        at: endPoint].  
    self drawHand: model time value. "Draw hand, getting seconds from the application model."
```

Note that we used addition to calculate theta because it is faster than multiplication. The low-level drawing method `draw:` displays the hand by drawing a straight line from the center of the face to the endpoint at an angle given by the time:

draw: integerSeconds

```
"Draw hand in position corresponding to seconds in integerSeconds and update input field."  
| endPoint |  
endPoint := center + (Point r: radius1 theta: integerSeconds * const1 - const2). "Angle corrected by 90°."  
self graphicsContext displayLineFrom: center to: endPoint.  
model time value: integerSeconds "Display digital value in input field."
```

Method `erase:` is very similar and the only differences are that it changes the paint of the graphics context to that of the background (to make the old hand invisible) and that it does not rewrite the input field. We found how to get the background color of a widget in the Cookbook.

erase: anInteger

```
"Erase hand in position corresponding to seconds in anInteger."  
| backgroundColor endPoint |  
    backgroundColor := (model builder componentAt: #watch) lookPreferences backgroundColor.  
    endPoint := center + (Point r: radius1 theta: anInteger * const1 - const2).  
    (gc := self graphicsContext)  
        paint: backgroundColor;  
        displayLineFrom: center to: endPoint
```

Note that if we were sure that we always wanted stopwatch background to have the same color, for example white, we could assign it directly as `ColorValue white`. The more general approach that we have used will work even if we change the background color property of the widget.

As we already said, the high level drawing and resetting is done by `move:` and `reset:`. The `move:` method first erases the old hand and then displays the new hand in the new position:

move: anInteger

```
"Move' hand to position defined by anInteger."  
    self erase: anInteger - 1;  
    draw: anInteger
```

The `reset:` method erases the hand at the old position (argument) and redraws it at 0 seconds as follows:

reset: anInteger

```
"Reset hand from position defined by anInteger to start position."  
    self eraseHand: anInteger;  
    drawHand: 0
```

The only remaining WatchView method is `update:`. This method is executed when the model sends itself `changed:` and this happens when the user clicks *Reset*, and inside the infinite loop when time increments. Each of these two behaviors is different and we must thus use `changed:` with a symbolic argument to trigger the proper response. We must also include an argument representing the time from which the stopwatch is being reset (in the case of *reset:*) or the new time in regular operation (method *move:*). If we use `#reset:` or `#move:` as the symbolic argument, the definition can be as simple as follows:

update: aSymbol with: integerSeconds

```
"Update the hand in a way appropriate for the current context represented by aSymbol."  
self perform: aSymbol with: integerSeconds
```

This will require that the model must send `changed:with:` with arguments `#reset:` or `#move:`, and time.

Class Stopwatch

During the start up of the application, we must initialize the *View* variable to a new instance of WatchView, set its model, and initialize the input field aspect variable.

initialize

```
watch := WatchView new.  
watch model: self.  
time := 0 asValue          "The type property of the input field is number."
```

We have already mentioned that when the builder is finished building the window, we must ask the subview to initialize its parameters in `postBuildWith:`:

postBuildWith: aBuilder

```
watch setParameters
```

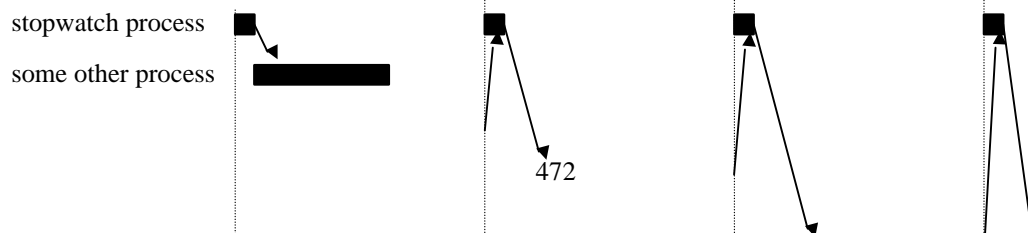
where `setParameters` is the WatchView method defined above. Except for accessing methods, it now only remains to write action button methods. The *Start* button opens an infinite loop in which it creates a one-second delay, updates time, and sends itself the `changed:with:` message with appropriate parameters. This is easy except for two things: we must be able to stop the loop by clicking the *Stop* button, and it must be possible to run another program when the stopwatch is active but not running. In the second case, the stopwatch program must wrestle control back when it needs to update.

In essence, what we need is to package the iteration that triggers the watch motion as a separate program, store access to this separate program in an instance variable, and use this instance variable to 'kill' the program from the *Stop* action method. Separately packaged and independently running programs are called *processes* (more on this shortly), and we will thus use the name *process* for the name of the instance variable referring to this separately running block. The whole definition of *start* is as follows:

start

```
"User clicked Start. Start a new separately running unending process and create a reference to it."  
process := [(Delay forSeconds: 1) wait.  
            time value: time value + 1.  
            self changed: #move: with: time value] repeat] fork
```

As you can see, VisualWorks' style of creating a process is to send the `fork` message to a block whose contents is the program corresponding to the process. This schedules the block for evaluation (more on this later) and makes it accessible as a process. Our particular block (*process*) runs only long enough to execute the `Delay` statement which stops the process for one second - allowing any other waiting independent program (*process*) to start running (Figure 13.2). (This description of the operation of processes is simplified and we will give a more accurate description later.)



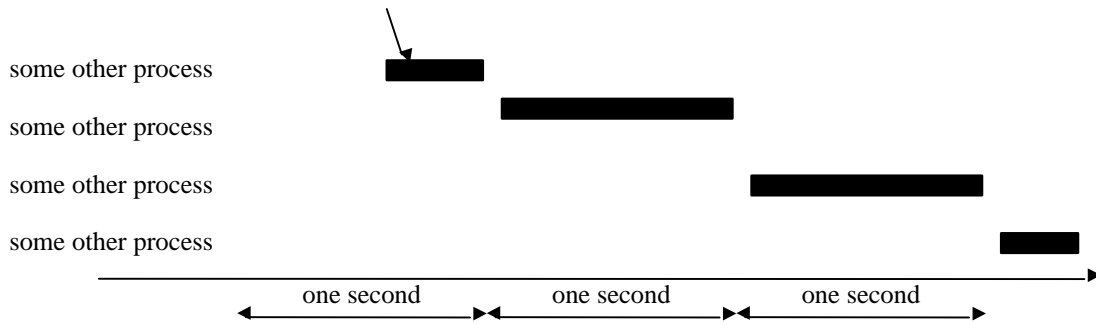


Figure 13.2. Simplified behavior of processes: Stopwatch runs until stopped by Delay. At this point another process can start execution and the stopwatch resumes when the one second delay expires.

The *Stop* button stops the infinite iteration process started by the *Start* button by sending it the terminate message:

stop

"Terminate the process implementing continuous stopwatch operation."
 process terminate

By the way, *stop* does not send *changed:with:* because it does not have any effect on the display - it just freezes the iteration. Action method *reset:*, on the other hand, sends *changed:with:* with *#reset:* and the current value of time because it must erase and redisplay the hand. The time argument is used to determine where the hand to be erased is:

reset

"Use dependency to reset stopwatch display to 0."
 self changed: #reset: with: time valu

where the 0 argument of *with:* is irrelevant because the message eventually executes *reset* in *WatchView* which resets to 0 anyway.

The program is now complete and ready to run! It works almost perfectly and allows us to execute short messages while the iteration is delayed. There is, however, one problem: If we start executing a longer message while the stopwatch runs, the display will freeze and continue incrementing only when the other program stops. To check this out, start the stopwatch, type

[3 factorial] repeat

into a *Workspace*, and execute it. The stopwatch will complete its current display, the factorial loop will jump in, and the stopwatch will never again get a chance to update. This behavior violates the part of the specification that states that execution of another program should not affect the stopwatch. Let's explore what is happening and how we can correct it.

When we start the stopwatch, the block

```
[(Delay forSeconds: 1) wait.  
time value: time value + 1.  
self changed: #move: with: time value]
```

starts executing and stops when it reaches

```
(Delay forSeconds: 1) wait
```

At this point, another program (such as our factorial loop - call it X) can take over. If this happens, we now have two programs competing for the CPU - the stopwatch, and program X. In computing terminology, these two independent programs are processes.

Roughly speaking, Smalltalk processes are designed in such a way that if one process runs, all other processes must wait until it is completed.¹ This arrangement is acceptable in most but not in all situations and our case is one of the exceptions: Since the stopwatch must not get behind, its process must be able to execute whenever it needs to update time and this means that the stopwatch process must have priority over other processes. If another process with lower priority is executing when stopwatch needs to update, the stopwatch process must assume control and execute. This is indeed possible because VisualWorks implements the concept of process priority and allows us to assign any one of 100 different priorities to any process.

With the concept of *process priority*, we can describe the operation of Smalltalk processes more accurately as follows: A process runs to completion, unless another process with higher priority wants to gain control. If this happens, the higher priority process becomes active and the lower priority process yields and waits until the higher priority process terminates or yields control. (In the stopwatch infinite loop, this happens when the process executes the Delay expression). This principle is illustrated in Figure 13.3.

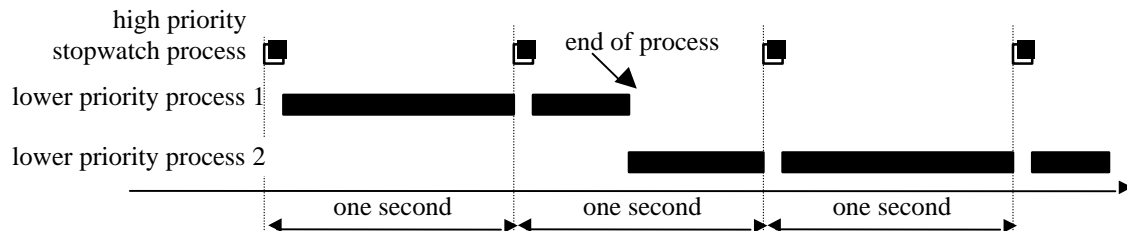


Figure 13.3. Processes and priorities: When a high priority process yields, lower priority processes can start execution. When a higher priority process requires processing, lower priority processes yield.

As we said, VisualWorks distinguishes 100 different levels of priority. Priority 1 is the lowest and 100 the highest. To maintain consistency with previous releases and to provide protection from changes of numbering in new releases, some priorities used by the system have names and can be assigned by sending the appropriate message to Processor. These named priority levels should be used whenever possible and their names and uses are as follows:

Priority number	name (method)	Purpose
100	timingPriority	Processes dependent on real time.
98	highIOPriority	Critical I/O such as network input.
90	lowIOPriority	Normal I/O such as keyboard input.
70	userInterruptPriority	High priority user tasks such as window management.
50	userSchedulingPriority	Normal user programs – default assigned by VisualWorks.
30	userBackgroundPriority	Background user processes.
10	systemBackgroundPriority	System background processes.

As an example, priority 50 can be assigned as

Processor userSchedulingPriority

although this particular priority normally does not have to be assigned because it is the default assigned when a user program starts executing. A user program with higher than normal priority - such as our stopwatch which should be able to interrupt other user programs - should thus run at *user interrupt priority* or at *timing priority* if the displayed time is really critical and must not suffer from any interruptions at all. We will run our stopwatch process at *user interrupt priority*.

¹ Process designs that do not allow other processes to run until the running process stops are called *non-pre-emptive*. In spite frequent statements to the contrary, we will see that VisualWorks processes are pre-emptive because a running process can be interrupted by a higher priority process.

In our modified start method, we want to create the stopwatch process, assign priority to it, and schedule it immediately. The simplest way to do this is to use `forkAt: aPriority` instead of `fork` as follows:

```
forkAt: Processor userInterruptPriority
```

The only change in our program is thus in process creation which now becomes

```
start
```

```
"Start a new separately running unending process and create a reference to it."
```

```
process := [Delay forSeconds: 1.
```

```
time value: time value + 1.
```

```
self changed: #move: with: time value] repeat] forkAt: Processor userInterruptPriority
```

When you now start the stopwatch and execute, for example,

```
[Transcript show: 'test'; cr] repeat
```

the stopwatch will continue running without interruption and the Transcript will continue printing, being interrupted (imperceptibly) by the higher priority stopwatch whenever its Delay expires (Figure 13.4).

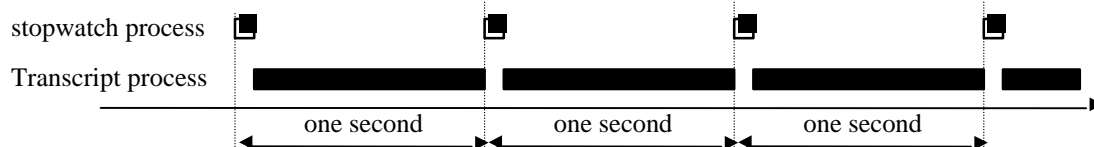


Figure 13.4. Higher priority stopwatch process taking turns with a lower priority process.

After this introduction to processes, we are now ready to take a closer look at their role and operation in VisualWorks.

Processes

Let's summarize what we have learned about Smalltalk processes so far: A process is a standalone thread of execution that can run on the same CPU and in the same Smalltalk image independently of other Smalltalk programs. Processes can be assigned priorities between 1 and 100 and at any given time, the highest priority process that is ready for execution runs. If a higher priority process requires processing, the running lower priority process is interrupted and its execution resumes when its priority becomes the highest waiting priority.

To create a process, send a process creation message to a block. We have introduced two process creation messages - `fork` and `forkAt:`. The first creates a process with the same priority as its parent process (the process from which it was forked) and starts its execution. The second assigns the process the specified priority and starts its execution if appropriate.

Creation of Smalltalk processes using `fork` essentially evaluates a block and message `fork` is thus quite similar to message `value`.² There is, however, an essential difference the two messages: The `value` message immediately evaluates its block and the message following the block must wait until block evaluation is finished. As an example,

```
[3 timesRepeat: [Transcript show: 'test']] value.
```

```
Transcript show: 'end of test'
```

```
prints
```

```
test
```

² Just as there are variations of `value` that send one or more arguments to a block, there is also a process creation message that allows forking a process with any number of arguments

```
test  
test  
end of test
```

This is not so for forking processes. As an example,

```
[3 timesRepeat:  
  [Transcript show: 'test']] forkAt: Processor userBackgroundPriority.  
Transcript show: 'end of test'
```

creates a *low priority* process, delays its execution until the parent process (the encompassing program) ends execution, executes the last line (thus ending the parent process), and if there are no other higher priority processes waiting, execution now returns to the suspended lower priority process and executes it. The code will thus print

```
end of test  
test  
test  
test
```

in the Transcript. In fact, we will get the same result even if the new process does *not* have lower priority than the parent process as in

```
[10 timesRepeat:  
  [Transcript show: 'test']] forkAt  
Transcript show 'end of test'
```

because the new process will be placed in a queue behind the parent process.

The only fundamental question that remains unanswered is who manages the processes and their priorities. Implementing process management is the role of class `ProcessorScheduler` which schedules the order in which the processor (the CPU executing the image) executes all existent processes. Since a single Smalltalk image needs only one scheduler, an image has only one instance of `ProcessorScheduler` and its name is `Processor`. `Processor` is a global variable like `Smalltalk`.

To close this section, we will now examine how `Processor` handles processes starting with a look at class `Process`. Class `Process` has several instance variables but for our purpose the most important ones are `priority` (an Integer between 1 and 100, naturally), and `myList`. Variable `priority` is, of course, process priority, and `myList` is a list holding all processes of the same priority in the order in which they will be activated after this process. When a new process with this priority is created, it is added to the end of this queue. This is why the forked process in our last example executed after its parent.

Normally, the first process in the queue must execute to completion before the process following it in the queue can start executing. One can, however, suspend a process (blocking its operation until it gets a receive message) or send it to the back of the queue by executing

`Processor yield`

To force a process that has not yet terminated to stop and to remove it from the queue, send it the message `terminate`. This is what we did in our stopwatch program - the stopwatch process runs in an infinite loop and the only way to stop it is to terminate it. Finally, for each priority, `Processor` (the single instance of `ProcessorScheduler`) keeps an array holding all process queues and uses it to determine which process to execute next.

This completes our introduction to processes and we will give further examples of their use and explain additional concepts in the remaining sections of this chapter. One important point that we have not covered at all is the coordination of processes. The point is that processes sometimes depend on one another and one cannot proceed before another completes some operation. As an example from the real world, consider the following somewhat unusual arrangement of traffic lights at a crossing of a road and a railroad designed to prevent the possibility of a collision: When a train or a truck arrive at the crossing, they change

the other vehicle's traffic lights to red to prevent collision. Upon leaving the crossing, the vehicle changes the other vehicle's traffic lights back to green. We will write a computer simulation of this arrangement in Section 13.3 and show how the desired coordination can be achieved by using the concept of a semaphore.

Main lessons learned:

- A process is a block of code – a program that shares the CPU processor and other resources with other programs under the control of a processor scheduler.
- Non-pre-emptive process scheduling lets a process run until it terminates or until it suspends its execution.
- Smalltalk processes are instances of class `Process`; they are pre-emptive because an available higher priority process interrupts a lower priority one..
- Three important `Process` messages are `suspend`, `resume`, and `terminate`. Message `suspend` takes its receiver out of the queue of runnable processes but does not destroy it. Message `resume` puts a suspended process at the end of the queue of runnable processes, and shuts down and destroys the process.
- Smalltalk processes have priorities between 1 (the lowest) and 100. By default, user processes run at priority 50, I/O processes such as mouse and keyboard events run at priority 90, real-time processes run at priority 100, and system background processes run at priority 10.
- To create a process, send `fork` or `forkAt: priority` to a block. This puts the new process at the end of the queue of runnable processes of the same priority and schedules it for execution.
- Smalltalk processes are managed by `Processor`, the single instance of class `ProcessScheduler`.
- `Processor` always selects for execution the first process in the highest priority non-empty queue of runnable processes.

Exercises

1. In our `reset` method, we used `reset: time value`. Shouldn't this be `reset: 0`?
2. Modify the stopwatch program to eliminate any effect of clicking *Start* or *Reset* while the stopwatch is running.
3. Add numeric time marks next to 0, 15, 30, and 45 second ticks.
4. Reimplement the stopwatch by creating an array of 60 hands, one for each possible hand position and each capable of drawing itself in a specified color. Then implement drawing and erasing by redrawing the current hand object in the background color and displaying the new one in the foreground color. Comment on the relative speed of both approaches.
5. Generalize the stop watch design by letting the creation method specify both the foreground and background colors.
6. Develop a digital version of a minute/hour watch.
7. Develop a `Watch` application with an analog face for hours and minutes.
8. Combine `StopWatch` and `Watch` into one application showing hours, minutes, and seconds.
9. Create other user interfaces for digital or analog hours, minutes, and seconds, and allow the user to select any desired combination.
10. Combine stopwatch, watch, and calendar into a new time machine application/widget.
11. Redefine the background paint of the stopwatch to a nice inconspicuous *pattern*.
12. Assume that an application can get blocked because a process that it is running can be suspended. Assume that the user interface contains an action button designed to restart the application. Since the button is a part of the application, the button cannot be used to unlock the process. True or false?
13. Assume that a forked off process has higher priority than the parent process. Will the parent process run to completion before the child process starts executing?

13.2 Alarm Tool

After implementing a stopwatch, the obvious next candidate in our exploration of time machines is an alarm clock. Since a flexible implementation of automated alarms opens some interesting new questions and helps to explore the concept of processes, we will dedicate this section to the design of a simple application allowing the user to create, edit, and delete any number of alarm notifications that pop up on the screen when their time arrives.

Specification

Develop a tool for creating, modifying, and deleting alarm events ('alarms', for brief), each characterized by expiry time and a brief description. The user can create any number of alarms at any time and in any order, and they will mature in the order of their expiry time, independently of the order in which they were created. When an alarm matures, it opens a notifier displaying its description. If two or more alarms mature at the same time, their notifiers are displayed in the order in which they were created. The operation of alarms must not interfere with user programs running at default priority, and operation of user default priority programs must not interfere with the alarms.

The desired user interface is as in Figure 13.5 and its behavior is as follows: The list of unexpired alarms has a pop up menu with command *add*. When an alarm is selected in the list, the pop up menu displays additional commands *edit* and *remove*. When the user selects *add*, the program opens the form shown on the right and when the user completes the form and clicks *Accept*, the new alarm is added to the list at a place corresponding to its maturity time. The window opens with current hours and minutes and does not accept time earlier than Time now. Clicking *Cancel* closes the form and aborts the operation. Editing uses the same form but the form opens with the parameters of the selected alarm. An attempt to remove an alarm elicits confirmation from the user.

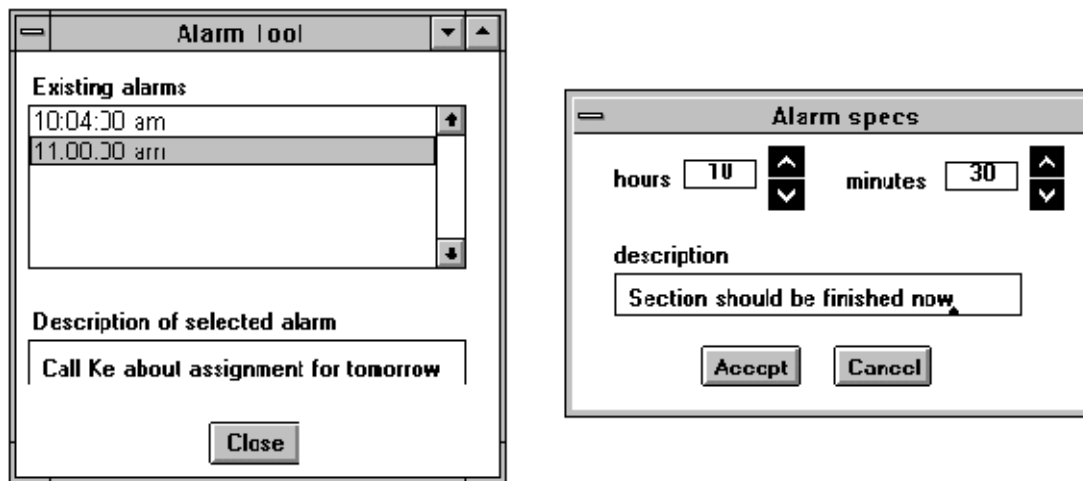


Figure 13.5. Main Alarm Tool window (left), form for creating and editing alarms (right).

The first iteration of design and implementation described below is intended as a test of the concept and implements the required functionality in a minimal way and further expansion is proposed as a series of exercises. The reason for this progressive enlargement of functionality is that the problem opens unfamiliar questions and we don't want to get bogged down in too many new questions at once.

Scenarios

For lack of space, we will leave out the more obvious scenarios and restrict ourselves to the more challenging ones.

Scenario 1: User adds an alarm to a non-empty list

1. User clicks *add* in the list pop up menu.

2. *Program* opens form.
3. *User* enters information and clicks *Accept*.
4. *Program* closes form, inserts new alarm in the list, and displays the updated list in the main window.

Scenario 2: User edits an alarm in the list

1. *User* selects an alarm and clicks *edit* in the list pop up menu.
2. *Program* opens the form, showing all parameters of the selected alarm. All fields are editable.
3. *User* edits the form and clicks *Accept*.
4. *Program* closes the form and changes the list to show the new alarm.

Scenario 3: User deletes an alarm

5. *User* selects an alarm and clicks *remove* in the list pop up menu.
6. *Program* displays a confirmation dialog.
7. *User* confirms.
8. *Program* deletes the alarm and changes the list to show the new alarm.

Context Diagram

The specification is so simple that there is no need to draw the Context Diagram.

Preliminary design

This application can be implemented with a single class - the application model (to be called AlarmTool). We don't even need to develop custom view classes because the user interface uses only standard widgets and windows. For cleaner design, however, we will add a class called Alarm to represent individual alarm objects. The preliminary description of our two classes is as follows:

- AlarmTool manages the Alarm Tool user interface, allows the user to create, edit, and delete alarms, and schedules and terminates alarms.
- Alarm holds all information about a single alarm.

Class level scenarios

Scenario 1: User adds an alarm to a non-empty

1. User clicks *add* in the list pop up menu.
2. AlarmTool opens a form (a modal dialog window) with default parameters.
3. User enters information and clicks *Accept*.
4. AlarmTool closes the form.
5. AlarmTool creates new alarm process with specified parameters and schedules it.
6. AlarmTool creates new Alarm with maturation time, description, and process created in the previous step.
7. AlarmTool adds the new Alarm to its collection and displays the new collection list ordered by alarm maturation time.

Scenario 2: User edits an alarm in the list

1. User selects an alarm and clicks *edit* in the list pop up menu.
2. AlarmTool terminates the process associated with the selected Alarm.
3. AlarmTool opens a form with parameters of the selected alarm.
4. User edits the form and clicks *Accept*.
5. AlarmTool closes the form.
6. AlarmTool deletes the selected alarm from its collection.
7. AlarmTool creates a new alarm process with specified parameters and schedules it.
8. AlarmTool creates a new Alarm with maturation time, description, and process created in the previous step.
9. AlarmTool adds the new Alarm to its collection and displays the new list.

Scenario 3 is similar to Scenario 2 and we leave it as an exercise.

Preliminary specification of class responsibilities

The class scenarios and the implied behaviors suggest the following responsibilities:

Alarm

Responsibilities

- Know and provide access to maturation time, description, and scheduled process.

Collaborators

AlarmTool

Responsibilities

- Create and manage user interface including the main window and the form.
- Maintain list of alarms in order of maturation.
- Schedule alarms according to their maturation times.

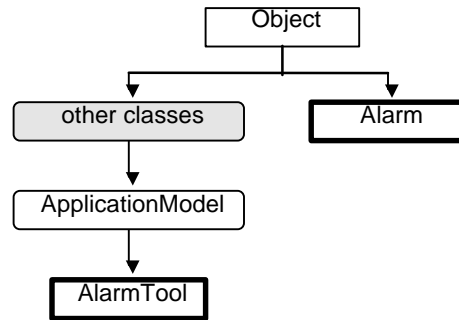
Collaborators

Alarm

Design Refinement

Class Hierarchy

AlarmTool is an application model, and its superclass is therefore ApplicationModel. Alarm does not have any relatives in the class library and it will thus be a subclass of Object (Figure 13.6).



13.6. Hierarchy Diagram.

Specification Refinement

Alarm

This class is a simple data model and requires only accessing methods. There is no need for further details.

AlarmTool

The first two groups of responsibilities (user interface and management of alarm list) are rather routine and we will thus start by examining how to implement scheduling, maturation, and expiry of alarms.

Each alarm is a simple process independent of others. Because of the requirement that an alarm be independent of other user programs, alarm processes must run at user interrupt priority. We will use Delay to schedule an alarm for maturation and proper action. The question is whether this will ensure that alarms will indeed mature in the correct order. The point is that alarms may be added in arbitrary order independent of the order of their maturation, and that the process scheduler activates processes in the order in which they occur in the queue which is normally the order in which they were added. It seems that our situation is peculiar in that all alarm processes immediately delay themselves and wake up only when their Delay expires. As a consequence, alarm processes automatically schedule themselves in the correct order.

To test this hypothesis, we will write a code fragment to simulate a mismatch between the desired order of execution of alarms and the order in which they are created by the following code fragment:

```
|seconds|
"Create an array of alarm maturation times."
seconds := #(7 4 9).
" Use these maturation times to create alarm processes and schedule them in the given order."
1 to: seconds size do:
[:index | | block |
    block := [(Delay forMilliseconds: (seconds at: index) * 1000) wait.
    Dialog warn: 'Alarm number: ', index printString].
    block fork]
```

We expect that the alarm created second will mature first, the alarm created first will mature next, and the alarm added last will also mature last. And this is indeed what happens. The basic idea thus works - but there are two little twists that we must examine next:

1. What if the user edits the maturation time or the description of a scheduled alarm before it matures?
2. What if the user deletes a scheduled immature alarm?

Our view of alarm editing is that editing an alarm is equivalent to deleting an existing alarm and adding a new one. Since we have just tested how to add a new alarm at any time, the only remaining question is deletion of an existing alarm.

To delete an existing alarm, we must terminate its corresponding process and remove the alarm from the list. Quite frankly now, we don't yet have enough experience with processes to satisfy ourselves with this formulation - we are not confident that this strategy will really work. We will thus create a skeleton of AlarmTool implementing just this part of its behavior and test that it works before we proceed to the design of the user interface and the rest. Our implementation may have little to do with our definitive implementation because the user interface will eventually be totally different and there will be some additional requirements, but that's OK. We only jump into the implementation phase for a moment to test the concept, and return to the rest of design later. Don't feel bad about this violation of orderly development, it is quite normal. We cannot continue designing responsibly if we don't know whether an essential part of our construction is valid or not and this detour is thus not only legitimate but even necessary. It also demonstrates that realistic development is loosely structured.

Partial and preliminary implementation of AlarmTool

For our test, AlarmTool needs only one instance variable (alarms), initialized to a SortedCollection of Alarm instances sorted by maturation time. (Since this is a test class, we called it AlarmToolTest.) According to our analysis, we only need two methods to handle alarm management - one for adding, and one for removing. After the user enters time and comment, the *add* operation creates and schedules a new process, combines it with time and description into an Alarm, and adds it to alarms. For the time being, we replaced Alarm with a three-element array and implemented the idea as follows:

addAlarm: anArray

"Create and schedule a process corresponding to alarm (represented by anArray with time and description), and the complete alarm information to alarm collection."

```
| process |  
process := self scheduleAlarmProcessWith: anArray.  
alarms add: (anArray copyWith: process) "Concatenate 2-element array into 3-element array."
```

where the scheduling of the process is performed as follows:

scheduleAlarmProcessWith: anArray

"Create and schedule a process with parameters described by anArray."

```
| block |  
block := [:seconds :comment |  
    (Delay forMilliseconds: seconds * 1000) wait.  
    alarms removeFirst. "Time expired, remove yourself from the list."  
    Dialog warn: 'Alarm: ' , comment].  
^ (block newProcessWithArguments: anArray)  
priority: Processor userInterruptPriority; resume
```

where newProcessWithArguments: is a message defined in BlockClosure that constructs a process from a block with arguments, in our case time and comment. Since VisualWorks does not have a method to create a process over such a block *and* schedule it in a single step, we created the process without scheduling it and sent it resume to schedule it. To test our ideas we now used the fragment

```
| alarm alarms |  
"Create an instance of the application."  
alarm := AlarmToolTest new.  
"Create suitable alarm descriptions."  
alarms := #(5 'number 1') #(2 'number 2') #(7 'number 3')).  
"Add and schedule these alarms one-by-one."  
alarms do: [:anAlarm | alarm addAlarm: anAlarm]
```


This should open alarm notifiers in the order 'number 2', 'number 1', and 'number 3' - and it does. Everything is OK thus far.

Now for *alarm removal*. In the final implementation, the alarm will be obtained from the user interface and thus guaranteed to be an element of alarms. For our test, we must remove alarms by manipulating them very explicitly as in

```
| alarm alarms |
"Create an instance of the application."
alarm := AlarmToolTest new.
"Create suitable alarm descriptions."
alarms := #(#(6 'number 1') #(2 'number 2') #(10 'number 3')).
"Add and schedule these alarms one-by-one."
alarms do: [:anAlarm | alarm addAlarm: anAlarm].
"Remove and terminate an alarm."
alarm removeAlarm: #(6 'number 1').
"Add and schedule a new alarm."
alarm addAlarm: #(7 'number 4')
```

The *remove* operation must find the appropriate element in the collection, get its process component, and terminate it:

removeAlarm: anArray

```
"Remove alarm from collection and terminate its alarm process."
| anAlarm |
anAlarm := alarms detect: [:el | (el at: 1) = (anArray at: 1)].
(anAlarm at: 3) terminate.
alarms remove: anAlarm
```

With this method, we can now run the test to add, remove, and add processes listed above. We are extremely happy to report that everything works as desired: Notifiers 2, 4, and 3 open in this order and nothing else. However, we must also warn you about another crucial experience.

During the coding and testing, we made a few small mistakes and it turns out that mistakes that affect process scheduling and termination may have very serious consequences because they can affect the operation of the whole VisualWorks environment which is itself based on processes. We suggest that in situations such as these you use the following strategy: Write your code but before you run it, file it out. Then do the test. If a disaster occurs, interrupt or terminate your session and exit without saving if necessary, restart and file in your code, correct it, file out the 'correct' code, and repeat. This way, you will not lose your work or damage the image.

We are now confident that our approach works and we return to design to refine the description of class Alarm.

Back to design refinement

Our current description of AlarmTool is as follows:

AlarmTool

Responsibilities

- Provide and manage user interface including the main window and the form.
- Maintain list of alarms in order of maturation.
- Schedule alarms for maturation according to their maturation times.

Collaborators

Alarm

We will now explore the three responsibilities further and summarize our findings in a final class description.

Provide and manage user interface including the main window and the form

The user interface consists of two windows:

- *Main window.* This window does not need any special widgets but the list requires two different menus depending upon whether an item is selected in the alarm list or not. The list itself is interesting because it displays only one of the three components of each Alarm object, namely its time of maturation. This will be easy to achieve because the list widget displays its constituent objects by sending them `displayString` and we can redefine `displayString` in Alarm to return only the Time component. This will ensure the desired display, while selection in the list will return the whole Alarm object.
- *Form.* This is a dialog window rather than the usual application window and its only special features are as follows:
 - When the form opens on a new alarm, it shows the current time but no description. When the form opens on an existing alarm, it shows the alarm's current parameters.
 - Time control buttons must not allow time less than Time now.
 - Time controls are restricted to 0 - 23 hours and 0 to 59 minutes. We will design them so that when the user clicks ^ at the end of the scale, the time component (minutes or hours) will start wrapping around. As an example, incrementing 59 minutes will produce 0 minutes, and decrementing 0 minutes will give 59 minutes.

Maintain list of alarms in order of maturation.

We will keep Alarm instances in `alarms`, a `SortedCollection` sorted by maturation time.

Schedule alarms for maturation according to their maturation times.

A summary based on our earlier experimental findings is as follows:

- Adding a new alarm: Open an un-initialized form. If the user closes the form by *Accept*, schedule a process with time and description obtained from the form, create an Alarm with these parameters and the process, and add it to `alarms` and the list widget.
- Removing an alarm: Ask user for confirmation, extract the process part of the selected Alarm and terminate it, remove selection from `alarms` and the list widget.
- Editing an alarm. Extract the process from the selected Alarm. Open a form with existing time and description. When the user accepts, terminate the selected alarm and remove it from the list, schedule a new process and construct new Alarm object with new parameters, and add the new Alarm to `alarms`.

We can now write our final class descriptions.

Alarm: I provide access to all information about a single alarm.

Components: time (Time), description (String), process (Process).

Responsibilities

Collaborators

- Creation - with time, description, and process. Method `time:desc:process:`
- Accessing: time, description, and process
- Printing: return string representation with time only via method `displayString`

AlarmTool: I manage the Alarm Tool user interface, allow the user to create, edit, and delete alarms, and schedule and terminate alarms.

Components: `alarms` (`SortedCollection`), aspect variables of main window and alarm form.

Responsibilities

Collaborators

- Creation and initialization - as for an application model.
- User interface including the main window, the form, menus, etc.
 - Main window (`windowSpec`)
 - Form window (`form`)
 - Resources (*up* and *down* buttons)
 - Pop up menus for the list widget in the main window (add, add/edit/remove)
 - Menu methods
 - Add alarm: open form initialized to current time, get data on *Accept*, schedule new process, create new Alarm, update list; ignore on *Cancel*.

Alarm

- Method addAlarm.
- Remove alarm: terminate selection's process, update list. Method removeAlarm.
- Edit alarm: open form initialized to current parameters, get data on *Accept*, extract process from selection and terminate it, remove selection from alarms, schedule new process, create new Alarm, update list. Method editAlarm.
- Maintain list of alarms in order of maturation.

Implementation

We will now show details of implementation of selected methods in the order of classes and their specification.

Class Alarm

This is a simple domain class and its only interesting method is displayString which is used by the list widget in the main window. It extracts the time component and converts it to String:

displayString

"Returned time component for display in List."

^time printString

The creation method initializes all three instance variables:

time: aTime desc: aString process: aProcess

"Create fully initialized Alarm."

^(self new) time: aTime; description: aString; process: aProcess

Class AlarmTool

User interface specs

User interface creation is routine, except for the *up* and *down* buttons in the form window which use images created with the Image Tool. The corresponding action methods are as follows:

hoursInc

"User clicked to increment hour. Keep hours within limits and wrap around if necessary."

| h |

hours value: ((h := hours value + 1) > 23

ifTrue: [0]

ifFalse: [h])

and

hoursDec

"User clicked to decrement hour. Keep hours within limits and wrap around if necessary."

| h |

hours value: ((h := hours value - 1) < 0

ifTrue: [23]

ifFalse: [h])

for hour buttons, and the action methods for minute buttons are similar.

Initialization

The initialization process is standard; alarmList is the Aspect of the alarm list widget:

initialize

```
"Initialize alarms and aspect variables, define change behavior and initial pop up menu."  
alarms := SortedCollection sortBlock: [:x :y | x time < y time].  
alarmList := SelectionInList new.  
alarmList selectionIndexHolder onChangeSend: #changedAlarmSelection to: self.  
alarmListMenuHolder := self menuWithAdd asValue.  
description := " asValue
```

User interface behaviors

According to initialization, the following message is sent when the user makes a new alarm list selection:

changedAlarmSelection

```
"The list widget has a new selection. Update pop up menu and alarm description aspect variable."  
| selection |  
selection := self alarmList selection.  
self alarmListMenu value: (selection isNil  
    ifTrue: [self menuWithAdd]  
    ifFalse: [self menuWithAll]).  
description value: (selection isNil  
    ifTrue: [alarms isEmpty  
        ifTrue: ['']  
        ifFalse: ['Select alarm to see its description.']]  
    ifFalse: [selection description])
```

where description is the Aspect of the description widget. Method alarmListMenu was specified as the *Menu* property of the list widget when we painted the interface, and the menu building methods are as follows:

menuWithAdd

```
"No alarm is selected, only add is allowed."  
| mb |  
mb := MenuBuilder new.  
mb add: 'add' -> #addAlarm.  
^mb menu
```

and

menuWithAll

```
"An alarm is selected, all three commands are available."  
| mb |  
mb := MenuBuilder new.  
mb add: 'add' -> #addAlarm; add: 'edit' -> #editAlarm; add: 'remove' -> #removeAlarm.  
^mb menu
```

The critical operations of adding, editing, and deleting menus are all triggered by the messages specified in these methods and are implemented as follows:

removeAlarm

```
"To remove an alarm, terminate its process and remove it from the alarms collection and the list widget."  
| selection |  
selection := alarmList selection.  
selection process terminate.  
alarmList list: (alarms remove: selection; yourself)
```

Note that we have not included a confirmation dialog; this is left as an exercise.

Adding and editing have much in common (both open the dialog window, respond to *Accept* and *Cancel*, schedule a new process, and update the list) and factoring the shared behavior out simplifies their definition. We define the *add* operation as follows:

addAlarm

```
"Prepare parameters for form (current time, no description) and send doAddition to do the rest."  
self initializeFormForNewAlarm.  
self doAdd
```

where the critical work is done in

doAdd

```
"Shared part of add and edit. Open form as dialog, ignore on Cancel, obtain time from hours and minutes,  
schedule new process, update list."  
| process |  
(self openDialogInterface: #form)  
  ifFalse: [^self].  
time := Time fromSeconds: hours value * 3600 + (minutes value * 60).  
process := self scheduleAlarmProcessWith: (Array with: time with: descriptionInForm value).  
alarms add: (Alarm  
  time: time  
  desc: descriptionInForm value  
  process: process).  
alarmList list: alarms
```

where hours and minutes are Aspect variables of the corresponding input fields. This method is also used by the editing method:

editAlarm

```
"To execute edit, prepare parameters for form and send doAddition to do the rest."  
self initializeFormForExistingAlarm.  
self doAddition
```

Initialization of the form for a new alarm (see *addAlarm*) is done by

initializeFormForNewAlarm

```
"Prepare form aspect parameters for new alarm."  
self hours value: Time now hours.  
self minutes value: Time now minutes.  
self descriptionInForm value: "
```

and editing an existing alarm uses

initializeFormForExistingAlarm

```
"Prepare form aspect parameters for alarm selected in list."  
| selection |  
selection := alarmList selection.  
alarms remove: selection.  
selection process terminate.  
self hours value: selection time hours.  
self minutes value: selection time minutes.  
self description value: selection description
```

By the way, when you open your alarm tool, you may get strange values for starting hours and minutes. To correct this, change you VisualWorks time zone. We leave this as an exercise.

The only remaining action methods are for the *Close* button in the main window, and for *Accept* and *Cancel* in the form. The *Close* button closes the application in the usual way. *Accept* and *Cancel* use the standard behavior of dialog windows and use methods *accept* and *cancel* defined in *SimpleDialog*; these methods close the window and return true (on *Accept*) or false (on *Cancel*). We relied on this behavior in the first statement in method *doAdd* above.

This leaves *scheduleAlarmProcessWith:* which uses the data obtained from the form to create a new process with given maturation time and description, and schedules it. Its definition is

scheduleAlarmProcessWith: anArray

"Create and schedule a process with data obtained from the form and returned in anArray."

```
| block |
block := [:maturationTime :comment |
  (Delay forMilliseconds: maturationTime asSeconds - Time now asSeconds * 1000) wait.
  Dialog warn: ('Alarm at: ', Time now printString, '\', comment) withCRs].
^ (block newProcessWithArguments: anArray)
  priority: Processor userInterruptPriority; resume
```

When the process resumes after the prescribed delay, it open the notification window showing the alarm's time and description, and terminates, removing itself from the process queue.

Main lessons learned:

- Since processes are created from blocks and blocks may have arguments, processes may also be created with arguments using *newProcessWithArguments:*.
- Method *newProcessWithArguments:* creates but does not schedule a process. Scheduling is by message *resume*.

Exercises

1. Processes are expensive in terms of CPU requirements and having many of them will slow down VisualWorks. A better approach is to have a single process keeping all alarms in a list and handling them as needed. Reimplement Alarm Tool using this alternative approach.
2. Extend the alarm's notion of time to include date.
3. Add the following behaviors if they are not yet functional:
 - a. Set alarms remain functional even when Alarm Tool is closed.
 - b. Set alarms remain functional from session to session.
 - c. When the Alarm Tool is closed and then reopened later in the same session, non-expired alarms remain set and can be individually edited and modified.
 - d. Extend Exercise c across sessions.
4. Make it possible to specify alarms in relative terms such as 'in ten minutes'.
5. Add recurrent alarms of various kinds such as 'every ten minutes', 'every ten minutes until canceled by user', and 'every ten minutes, five times in a row.'
6. Add alarms that occur on a specified schedule.
7. According to the current specification, there is no restriction on alarm interference with other higher priority processes such as the stopwatch. Comment on the possibility of running alarms and the stopwatch at the same time, possible problems (if any), and possible remedies.
8. Add a digital clock to Alarm Tool.

13.3 Coordinating mutually dependent processes - Train Simulation

In many important situations, two or more processes are mutually dependent because they share some resource. In situations like these, conflicts may arise and processes require coordination. A typical

example of such a situation is printing on a shared network printer. A print job cannot be started while another job is printing and the processor managing the print queue must therefore wait for the printer to signal that it is ready before it sends another job.

A physical world example of the need to coordinate processes is the previously mentioned crossing of railroad track and a highway road: If a train is crossing, vehicles coming to the crossing must stop and this is normally implemented by a semaphore - a traffic light activated by the arriving train, and deactivated when the train leaves the crossing area. We will use this example to illustrate process coordination in Smalltalk. First, however, some background.

Semaphores

A Smalltalk semaphore is an analogy of a train crossing semaphore implemented as an instance of class `Semaphore`. The essential messages understood by semaphores include `new` (to create a new instance), `wait` (to indicate the desire to use the resource guarded by the semaphore and to turn the 'red light' on), and `signal` (to indicate that the resource is no longer needed and to turn the 'green light' on).

When two or more processes interest in the same resource, they use a shared `Semaphore`. Note that a `Semaphore` has no explicit connection to a process and behaves very much like a portable signaling device that can be carried around and placed to guard a resource anywhere the user wants.

When a process needs to use the resource guarded by a `Semaphore`, it sends `wait` to it. The `wait` message performs two functions: It first checks whether the 'light is green', in other words, whether the resource is available. If it is, it 'turns the light to red' to indicate that the resource is now in use and unavailable to other processes, and the process that sent it can continue executing its block. If the light is red, the `Semaphore` puts the process in a queue waiting for this `Semaphore` to turn green again.

When a process wants to indicate that it is finished using the protected resource, it sends the `Semaphore` the `signal` message. This message turns the light to green, and if there are any processes waiting at this `Semaphore`, the one at the head of the queue (the one that was the first to send `wait`) can resume. As we can anticipate from our knowledge of processes, the process will actually resume only when the currently active process gives up control by terminating, by suspending itself, or by being sent to the end of the queue by `Processor yield`. Note that the consequence of the described mechanism is that a process that issues `wait` can only proceed if the semaphore has received a `signal` message before.

The principles that we have just explained are summarized in Figure 13.7.

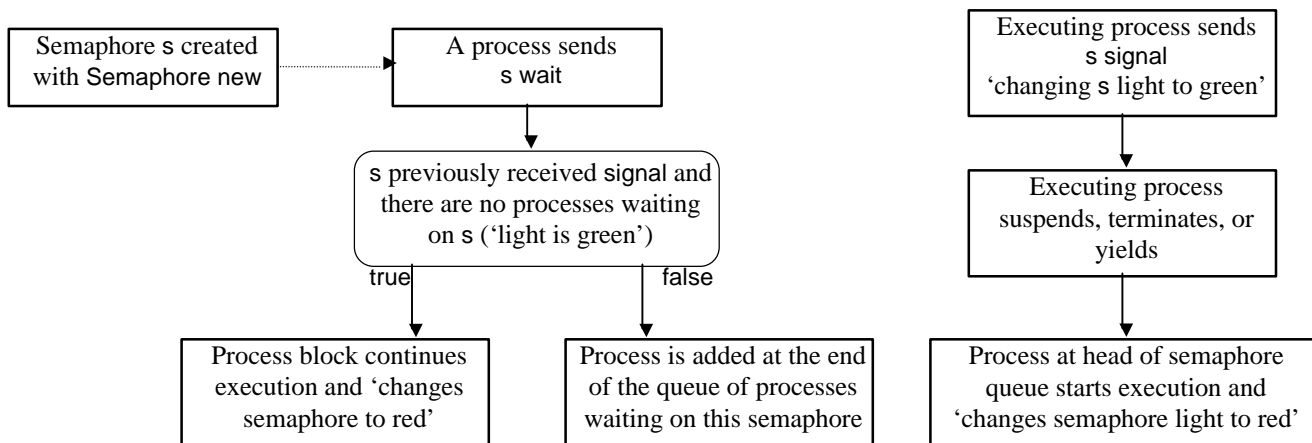


Figure 13.7. Operation of Semaphore. Any number of semaphores may exist at a time.

Train simulation - specification

After this introduction, we are now ready to give an example of the use of semaphores.

Problem: Develop a simulation of crossing truck and train tracks with the user interface in Figure13.8. At the beginning of the simulation, the truck is at the top of its track and the train at the left end of its track. When the user clicks *Go*, the train and the truck start moving at a constant speed, changing direction when they reach the end of the track. If a vehicle reaches the beginning of the crossing area (its own traffic light) and its semaphore light is green, it turns the other vehicle's semaphore lights to red. It then proceeds, moving in the usual way. When it reaches the end of its crossing area, it turns the lights on the other vehicle's track back to green. If the semaphore light on a vehicle's track is red when it arrives at the crossing, it stops and waits until the light turns green. Simulation continues until the user clicks *Stop* and may be restarted by clicking *Go* again.

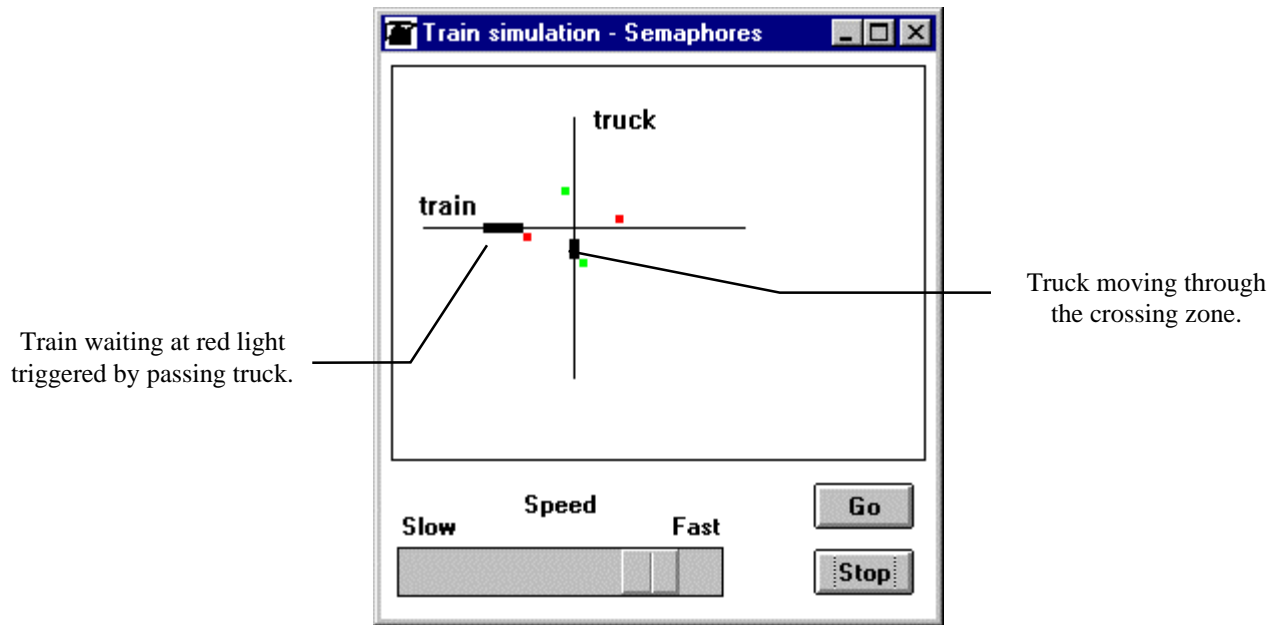


Figure 13.8. User interface for train simulation.

Scenarios

Scenario 1: Starting or restarting simulation

1. *User* clicks *Go*.
2. *Program* starts moving both vehicles from their current position. Motion is 'simultaneous'.

Scenario 2: Stopping simulation

1. *User* clicks *Stop*.
2. *Program* stops both vehicles in their current position.

Scenario 3: Changing speed

1. *User* moves the speed control slider to a new position.
2. *Program* changes speed setting for both vehicles and if the simulation is running, the new setting immediately takes effect.

Scenario 4: Train reaches the start of the crossing and its semaphore light is green

1. *Program* changes light on the truck side to red.
2. *Program* advances train to next position.

Scenario 5: Train reaches the start of the crossing and its semaphore light is red

1. *Program* stops the train at the light.
2. *Program* keeps advancing the truck until it reaches the end of the crossing zone.
3. *Program* turns train lights to green.
4. *Program* moves train and turns truck lights to red.

Scenario 6: Train reaches end of crossing

1. *Program* changes truck lights to green.

Scenario 7: Train reaches end of track

1. *Program* reverses direction of train motion.

Truck scenarios are analogous to train scenarios.

Preliminary Design

The domain objects include the train and the truck and we will leave it to the application model to take care of the simulation. It could be argued that the simulation object could be a part of the domain model and separate from the application model but we don't expect it to be complex. Finally, a layout view will implement the track layout subview but there is no active controller because the simulation subview does not allow any user interaction. Altogether, we need the following classes:

Train. Knows its position and direction of motion, knows how to calculate its next position, check that it reached the end of the track or the beginning or the end of its crossing, and how to turn around (change direction). Knows its simulation parameters such as its length (the length of the rectangle).

Truck. Same responsibilities as Train but truck-specific.

TrainSimulation. Application model. Contains the specification of the window and knows about the view class implementing the simulation subview. Knows global simulation parameters including start and length of train and truck tracks, and the limits of the crossing area (distance from crossing to lights). The *Go* action button is the heart of the simulation, *Stop* suspends simulation.

LayoutView. Display tracks, lights, and vehicles.

We will now examine the role of our classes in the previously listed scenarios.

Scenarios

Scenario 1: Starting or restarting simulation

1. *User* clicks *Go*.
2. *Program* starts moving both vehicles from their current position. Motion is 'simultaneous'.

This scenario is one of the essential components of the problem and before we start expanding it, we must have a conceptual model of the simulation. The problem invites the use of processes, one for each vehicle; after all, each vehicle is autonomous. The requirement that vehicle motion be 'simultaneous' means that the two processes must systematically take turns.

An important part of the motion of vehicles in our problem is the crossing and we must thus consider it in this context. The behavior of the crossing suggests the use of a semaphore shared by the two vehicles. When a vehicle reaches the crossing zone, it tells the semaphore about its desire to move and if the semaphore is ready (has been signaled), it allows the vehicle to proceed and change lights.

Another question is the relation of the train and truck *processes* to the Train and Truck *classes*. Our train process will use the Train domain object to perform all the calculations and trigger the display, and the

truck process will use the Truck class in a similar way. Process objects thus use domain objects but are separate from them. With this background, we can now expand Scenario 1.

Scenario 1: Starting or restarting simulation

1. *User* clicks *Go*.
2. TrainSimulation creates a train process and a truck process.
3. TrainSimulation starts the train process and the vehicle processes automatically start taking turns executing single step motion while respecting the semaphore at the crossing.

Scenario 2: Stopping simulation

1. *User* clicks *Stop*.
2. TrainSimulation terminates the two processes.

The immediate question is whether clicking *Stop* can actually freeze simulation. The point is that while the application is executing the simulation loop, other methods can only be accessed from the loop. Under these conditions, will the application be able to respond to the *Stop* button? The answer is that the *Stop* button will still be effective because our processes run at *UserSchedulingPriority* (the priority of their parent process - our simulation), whereas keyboard and mouse events are managed by a process running at *LowIOPriority* whose priority is higher. Clicking *Stop* will thus take precedence over simulation and activate the action method of the *Stop* button even though the train and the truck processes are running³.

Scenario 3: Changing speed

1. *User* moves the slider to a new position.
2. *Program* changes speed setting and if the simulation is running, the new setting immediately takes effect.

To control speed, we will end each simulation step with a *Delay* whose duration will be derived from the setting of the slider. For slow speed, the delay will be long, for fast speed, the delay will be short. Since slider control runs under *LowIOPriority*, the effect of dragging the slider marker to a new position immediately changes its value aspect. To control speed during simulation we must thus obtain the current slide position in each simulation step and use it to control the delay. (We will explain the simple rules for using a slider in the Implementation section.) We can now expand the scenario as follows:

Scenario 3: Changing speed

1. *User* moves the slider to a new position.
2. The active vehicle Process reads the new slider setting and uses it to create and schedule a *Delay* of appropriate duration.

Scenario 4: Train reaches the start of the crossing area and its semaphore lights are green

1. The Train object checks its current position and finds that it has just reached the start of the crossing zone. This test involves checking the current direction of motion, the position of the crossing, its width, and the length of the vehicle.
2. The Train object sends a wait message to the semaphore.
3. Since the semaphore is presumably *not* blocked, the train process is not suspended, wait turns the semaphore lights on the truck track to red, and the process advances the train to the next position.

Scenario 5: Train reaches the start of the crossing area and its semaphore lights are red

1. The Train object checks the current train position and finds that it has just reached the start of the crossing zone.
2. The Train object sends wait to the semaphore.
3. Since the semaphore *is* blocked, it suspends the train Process.

³ We now realize that we should have asked the question of how the user interface can take over from the application a long time ago - this principle is the essence of all interactive applications.

4. The truck which is moving through the crossing eventually leaves the crossing area, sends signal to the semaphore (Scenario 6), and suspends itself temporarily (see the alternation of train and truck processes in Scenario 1). The blocked train Process resumes, turns the semaphore lights on the truck track to red, and advances the train to the next position.

Scenario 6: Train reaches the end of the crossing area

1. The Train object recognizes the 'end-of-crossing' position, changes truck lights to green, and sends signal to the semaphore, unblocking a truck possibly waiting for green light.

Scenario 7: Train reaches end of track

1. The Train object recognizes the 'end-of-track' position. It changes its direction state which controls whether the next position is calculated by incrementing or decrementing the current position, and makes a move.

Preliminary object diagram

Except for the class hierarchy and possible abstract classes, we now have a good understanding of the mutual relationships of our classes. The corresponding diagram is in Figure 13.9.

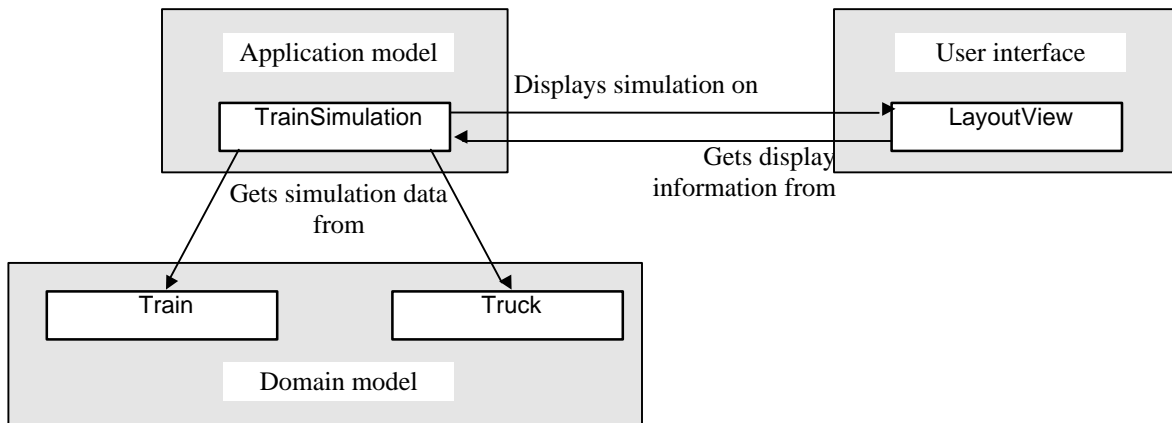


Figure 13.9. Preliminary Object Model of train simulation.

Design refinement

Class hierarchy

Domain objects Train and Truck have a lot in common: Both use the same principle to move, and both must know how to check crossing boundaries and end of track, and how to turn. We will thus create an abstract class called Vehicle implementing as much of the shared functionality as possible, and delegate vehicle-specific details to Train and Truck. TrainSimulation is our application model and a subclass of ApplicationModel, LayoutView is a subclass of View. The resulting class hierarchy is as in Figure 13.10.

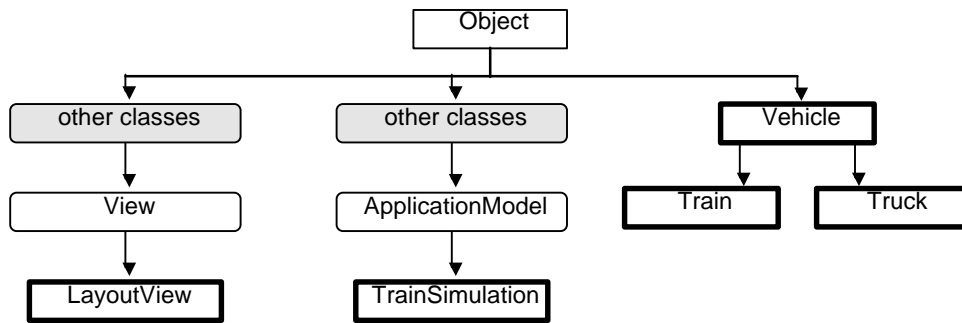


Figure 13.10. Class Hierarchy Diagram with classes to be designed shown as heavy rectangles.

Refined class descriptions

We now understand enough of the behaviors of our classes to be able to start refining class descriptions:

Domain classes

Vehicle. Abstract superclass of domain classes Train and Truck. Factors out shared behavior and knowledge such as vehicle position and direction of motion, calculation of the next position, testing for the end of the track or crossing, and turning.

Superclass: Object

Components: limit (distance from crossing to lights), position (distance of vehicle from start of track), direction (#incrementing or #decrementing position), vehicleLength (pixels), model (reference to the TrainSimulation object running the simulation).

Behaviors

- initialization - position and direction; method initialize
- moving - trigger display in new position, activate Delay
 - move - check position with respect to crossing, send wait or signal to semaphore and redisplay lights if necessary, check for end of track and turn if necessary, move one step
 - moveOneStep - calculate new position using direction information, display (ask vehicle for its display object), delay
 - semaphores: aColorValue - ask model to change color of semaphore lights
- testing - check whether entering or leaving crossing zone (enteringSemaphore, leavingSemaphore), check for end of track (endOfTrack)

Collaborators

TrainSimulation,
Train, Truck

Train, Truck

TrainSimulation

Vehicle-specific parts of these behaviors are left as subclass responsibilities for Train and Truck.

One note is in order about our design: The name of the variable model suggest dependency and a possible better implementation. We leave this question as an exercise.

Train. Concrete train vehicle. Implements train specific responsibilities and specializes those inherited from Vehicle.

Superclass: Vehicle

Components: Inherited components

Behaviors

- initialization - execute inherited Vehicle initialization and initialize train specific track and vehicle parameters - method initialize

Collaborators

Truck. Same as Train but truck-specific.

Application model

Before we can write the description of TrainSimulation, we must refine our understanding of its critical task - the creation and nature of the processes that represent the train and the truck. We have already decided that each of these processes will be implemented as an infinite loop that asks the corresponding domain object (a Train or a Truck) to make a move (including redisplay), and then somehow yields to the other process. Our first hunch is to do something like

```
trainProcess := [ [train move.  
                  trainProcess suspend.  
                  truckProcess resume] repeat] fork.  
truckProcess := [ [truck move.  
                  truckProcess trainProcess suspend.  
                  trainProcess resume] repeat] fork
```

In this implementation, a vehicle process performs a one-step move, resumes the other process (currently suspended), and suspends itself, activating the resumed process. Unfortunately, this approach will not work: Assume that the truck is in the middle of the crossing zone when the train reaches the semaphore light. Its move operation issues

semaphore wait

and is placed in the semaphore waiting queue because the light is red. The truck makes one step, suspends itself (see above), and waits for the train to resume it. However, the train is waiting for the truck to turn the semaphore to green – which it can't do because it cannot move. The train and truck processes become *deadlocked*. To solve this problem, we decide to rely entirely on the semaphore and code the processes as follows:

```
trainProcess := [[train move.  
                 Processor yield] repeat] fork.  
truckProcess := [[truck move.  
                 Processor yield] repeat] fork
```

To understand how this approach works, assume that none of the vehicles reached the crossing zone yet and that it is the train's turn to move. The following then happens:

1. The train makes a move and executes

Processor yield

asking Processor to put it at the end of the queue of runnable processes (processes that are not suspended or waiting for a semaphore). This leaves the truck at the head of the queue.

2. Since the truck is not waiting on the semaphore, it executes a move and asks Processor to put it at the end of the queue. This leaves the train at the head of the queue, the operation proceeds as described in Step 1, and so on.

Eventually, one of the vehicles (for example the train) reaches the beginning of the crossing zone. As a part of executing its move (see description of Train and Vehicle above), it sends

semaphore wait

turns the truck's light to red, and yields via Processor. If the truck is not yet in the crossing zone, it makes its move, yields to the train, and so on. If the train reaches the end of the crossing zone before the truck reaches its start, it turns truck's light to green and everything proceeds as in steps 1 and 2. If, however, the truck reaches the crossing zone while its semaphore is still red, its

semaphore wait

blocks the truck process. The truck process is now removed from the queue of runnable processes and the train is now the only process in that queue. The train makes a move, and yields. But since there is no other process in the queue of runnable processes, this does not stop it, it executes another move, and yields. If the did not reach the end of the crossing zone, the yield again does not change anything and the train makes another move (while the truck is still waiting on the semaphore). Eventually, the train reaches the end of the crossing zone, sends

semaphore signal

and this removes the truck from the semaphore queue and puts it in the queue of runnable processes. When the train now executes

Processor yield

and goes to the end of the queue, it now finds itself behind the truck which makes its move, yields, the train makes its move and yields, and so on.

With this background, we can now write a description of class TrainSimulation as follows:

TrainSimulation. Responsible for user interface and simulation.

Superclass: ApplicationModel

Components: delay (Integer - current value of delay between vehicle moves – controlled by slider), layout (View), semaphore (Semaphore), train (Train), truck (Truck), trainTrackLength (Integer), truckTrackLength (Integer), trainTrackStart (Point), truckTrackStart (Point), trainProcess (Process), truckProcess (Integer), speed (Integer), aspect variables.

Behaviors

- initialization - initialize instance variables, send signal to semaphore to get it ready to pass the first incoming vehicle; method initialize
- displaying - pass requests to display vehicle or change lights to LayoutView; methods display: and semaphores:for: (color and type of vehicle)
- actions - response to action buttons
 - go - create the two mutually yielding processes and start one of them
 - stop - terminate both processes

Collaborators

Train, Truck,
Vehicle,
LayoutView
LayoutView

Train, Truck

At this point, we realize that we should not allow the user to close the window and terminate the application if the processes are running. We thus add a `changeRequest` method that intercepts each attempt to close the window, checks whether there are any running processes (Boolean variable `oKToClose`), and if there are, warns the user and refuses to close the window.

User Interface

LayoutView. Display tracks, lights, and vehicles.

Superclass: View

Components: Parameters of layout geometry obtained from TrainSimulation – cached for greater efficiency.

Behaviors

- displaying - view, train and truck in new position, lights
 - displayOn: - the standard required display method
 - display Train or Truck upon direct request from domain objects via TrainSimulation; method display: with type of vehicle
 - display of semaphore lights; methods displayTrainSemaphores: and

Collaborators

LayoutView

displayTruckSemaphores: with a ColorValue argument

- controller specification - specify NoController in method defaultControllerClassr

Object Model Diagram

The only difference from the preliminary diagram is that we now know that Train and Truck have superclass Vehicle (Figure 13.11).

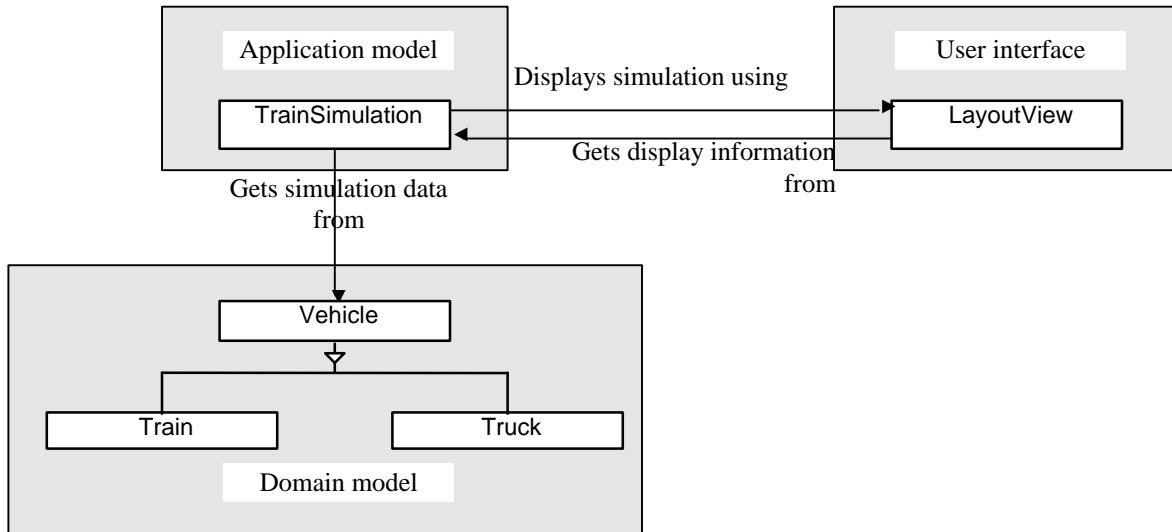


Figure 13.11. Final Object Model of train simulation.

Implementation

We will now show implementation details of several selected methods class by class.

Class Vehicle

Initialization is shared between the abstract vehicle and the concrete subclass. The shared part is as follows:

initialize

"Initialize parameters shared by Train and Truck. Leave the rest to Train and Truck classes."

position := 0.

direction := #incrementing

Most of the move operation is implemented in this abstract class and we split it into methods that perform tests for crossing limits and end of track, and a method that performs a one-step move. The principle is

move

```
"Test for crossing limit and end of track, make one-step move if appropriate."  
"Test whether we are at a crossing limit. If so, message semaphore and make one-step move."  
self comingToSemaphore  
    ifTrue: [model semaphore wait.  
            self semaphores: ColorValue red].  
self leavingSemaphore  
    ifTrue: [model semaphore signal.  
            self semaphores: ColorValue green].  
"Check for end of track and 'turn' if required."  
self endOfTrack ifTrue: [self turn]  
"Make a one-step move."  
self moveOneStep.
```

but to speed up execution, we perform one-step moves on each successful test and exit to avoid further tests:

move

```
"Test for crossing limit and end of track, make one-step move if appropriate."  
"Test whether we are at a crossing limit. If so, inform semaphore and a make one-step move."  
self comingToSemaphore  
    ifTrue: [model semaphore wait.  
            self semaphores: ColorValue red.  
            ^self moveOneStep].  
self leavingSemaphore  
    ifTrue: [model semaphore signal.  
            self semaphores: ColorValue green.  
            ^self moveOneStep].  
"Check for end of track and 'turn' if required."  
self endOfTrack ifTrue: [self turn]  
"No special situation - make a one-step move."  
self moveOneStep.
```

The move itself is performed by moveOneStep as follows:

moveOneStep

```
"Move one step in the current direction and pause briefly to provide the required control of speed."  
direction == #decrementing  
    ifTrue: [position := position - 1]  
    ifFalse: [position := position + 1].  
model display: self symbol.  
(Delay forMilliseconds: model delay) wait
```

We delegate display to TrainSimulation by asking it to display a vehicle identified by its symbol. The symbol method is defined in Train as

symbol

```
^#train
```

and similarly in Truck. The turn that occurs at the end of the track is implemented as follows:

turn

```
"Reached end of track, turn."  
direction == #incrementing  
    ifTrue: [direction := #decrementing]  
    ifFalse: [direction := #incrementing]
```

and the test methods are

comingToSemaphore


```
"Return true if I have just reached the start of the crossing zone."  
  ^ (direction == #incrementing and: [position = (crossing - limit - vehicleLength)])  
    or: [direction == #decrementing and: [position = (crossing + limit)]]
```

and

endOfTrack

```
"Return true if this is the end of the track. Remember that vehicle might have turned."  
  ^ (direction = #decrementing and: [position = 0])  
    or: [direction == #incrementing and: [position = (trackLength - vehicleLength)]]
```

The method that triggers redisplay of semaphore lights uses the same principle as `moveOneStep` and asks `TrainSimulation` to display semaphores controlled by the vehicle in the specified color:

semaphores: aColorValue

```
"Ask TrainSimulation to display semaphores controlled by vehicle identified by its symbol."  
  model semaphores: aColorValue for: self symbol
```

As you can see, `Vehicle` implements most of the functionality of its concrete subclasses.

Class Train

Initialization performs inherited initialization and sets initial values of train-specific parameters:

initialize

```
"Initialize vehicle parameters via super and Train specific values."  
  super initialize.  
  limit := model trainCrossingLimit.  
  vehicleLength := 20.  
  crossing := model truckTrackStart x - model trainTrackStart x.  
  trackLength := model trainTrackLength
```

The rest are accessing methods including `symbol` (listed above) and

crossing

```
"Calculate distance from start of track to intersection."  
  ^ model truckTrackStart x - model trainTrackStart x
```

Class Truck

All methods in class `Truck` are truck-specific variations on `Train` methods.

Class TrainSimulation

Initialization of `TrainSimulation` consists of setting the parameters of the layout, creating a `Train` and a `Truck`, creating a `Semaphore` and signaling it to get ready for the first train, creating and assigning the layout view, assigning initial simulation speed, and setting the variable that reflects the state of the train and truck processes and determines whether the application can be closed or not.

initialize

```
trainTrackLength := 160.  
truckTrackLength := 130.  
trainTrackStart := 15 @ 80.  
truckTrackStart := 90 @ 25.  
trainCrossingLimit := 25.  
truckCrossingLimit := 20.  
truck := Truck newOnModel: self.  
train := Train newOnModel: self.  
semaphore := Semaphore new.
```

```
semaphore signal.  
layout := LayoutView new.  
layout model: self.  
speed := 25 asValue.      "Aspect variable associated with the speed slider."  
oKToClose := true
```

The essential go action method is designed along the lines decided during Design Refinement:

go

"User clicked Go. Clear oKToClose to indicate that we have active processes and cannot close the application, create and schedule the train and truck processes."

```
OKToClose := false.  
trainProcess := [ [train move.  
                  Processor yield] repeat] fork.  
truckProcess := [ [truck move.  
                  Processor yield] repeat] fork
```

The *Stop* button terminates the two processes and sets oKToClose to indicate that the application can now be closed:

stop

"Terminate train and truck processes, sts OKToClose to True to indicate that the application may close."

```
truckProcess terminate.  
trainProcess terminate.  
OKToClose := true
```

Attempts to close the window send *changeRequest*. The method returns true when it is OK to close the application (when the vehicle processes are not running), and false otherwise:

changeRequest

"Decide whether it is OK to close the application."

```
oKToClose  
    ifFalse: [Dialog warn: Click Stop to stop running processes before closing.].  
^oKToClose
```

Class *TrainSimulation* also defines a group of methods responsible for display. In essence, they pass requests for display coming from vehicles to the layout view. They include

display: aSymbol

"Pass request to display a Train or a Truck to layout view."
layout display: aSymbol

and

semaphores: aColorValue for: aSymbol

"Pass request to display lights to the layout view."
layout displaySemaphoresFor: aSymbol inColor: aColorValue

Finally, the implementation of the slider. The slider is a simple widget whose value holder holds a number corresponding to the position of the marker of the slider bar. When the position of the marker changes, the value changes and when the value is changed programmatically, the marker moves to the corresponding position. All we need to do to implement a slider is to specify the start and the end values corresponding to the left and right endpoints using the Properties Tool, define the aspect variable (we called it *speed*), and define its initial value which we did in the initialization method above. When assigning the endpoint values, don't forget that the leftmost slider position represents the *slowest* speed which corresponds to the *largest* value of delay. The value of *speed* is used to calculate the delay

delay

^speed value

and both vehicles use this method to determine the length of their delay.

Class LayoutView

In addition to specifying NoController as the default controller class, the major responsibility of LayoutView is displaying. This responsibility can be divided into two tasks: Response to damage notification, and request for redisplay initiated from the domain model by vehicle motion. Response to damage notification is, as always, implemented by a displayOn: aGraphicsContext method. It draws the layout, the train and the truck, and the lights, and its definition as follows:

displayOn: aGraphicsContext

"Display tracks, vehicles, semaphores, and labels."

```
aGraphicsContext
  displayLineFrom: self trainStart to: self trainEnd;
  displayLineFrom: self truckStart to: self truckEnd;
  displayString: 'train' at: self trainStart x + 4 @ self trainStart y - 6;
  displayString: 'truck' at: self truckStart x + 4 @ self truckStart y + 6.
self displayTrainSemaphoresOn: aGraphicsContext withColor: model trainSemaphoreColor;
displayTruckSemaphoresOn: aGraphicsContext withColor: model truckSemaphoreColor;
displayTruck;
displayTrain
```

The display a vehicle is implemented as follows:

displayTrain

```
self graphicsContext displayRectangle: (0 @ 0 corner: self trainLength @ 5)
  at: self trainStart + (self trainPosition @ -2)
```

and the methods displaying the semaphores are explained below.

Vehicle requests for redrawing arrive from TrainSimulation but are triggered by the vehicles. They require redrawing of vehicles and semaphore lights. To redraw the vehicles, we can use one of two approaches. One is based on the view that we want to move a geometric object (the train rectangle) from one position to another. This high level approach deals with redrawing as a simple form of animation and to implement it, we can use methods follow:while:on: or moveTo:on:restoring: defined in class VisualComponent. We leave this approach as an exercise and use the low level perspective explained next.

Instead of thinking about moving a visual object, we can take the view that moving a rectangle horizontally or vertically by one pixel which requires only erasing the trailing edge and drawing a line in front of the leading edge. If the current position of the trailing edge corresponds to the crossing, we don't erase it because it also represents the track. To 'erase' the back of the vehicle, we redraw the corresponding line in white. After erasing the end of the rectangle, we must also redraw the piece of track that we erased in the process. This brings up the important fact that to display a point in VisualWorks, you must draw a one-pixel long line. The whole definition, as implemented for the train, is as follows:

displayTrainOn: aGraphicsContext

"Calculate the position of the trailing and leading edges and redraw them to simulate motion."

```
| length direction position oldEnd newStart |
length := model trainLength.
direction := model trainDirection.
position := model trainPosition.
direction == #incrementing
  ifTrue: [oldEnd := position - 1.
           newStart := oldEnd + length]
  ifFalse: [oldEnd := position + length + 1.
            newStart := position].
"Erase end of train except when on crossing."
oldEnd = (crossing x - trainStart x) ifFalse:
```

```
[aGraphicsContext paint: ColorValue white;
  displayLineFrom: self trainStart + (oldEnd @ -2)
    to: self trainStart + (oldEnd @ 2)].
aGraphicsContext paint: ColorValue black;
  "Draw leading edge of train."
  displayLineFrom: self trainStart + (newStart @ -2)
    to: self trainStart + (newStart @ 2);
  "Redraw erased part of track."
  displayLineFrom: self trainStart + (oldEnd @ 0)
    to: self trainStart + (position @ 0)
```

We had to correct the arithmetic expressions in this method several times to get the display to work properly, mainly because we did not bracket the binary messages + and @ and they executed differently from our expectations. This is a frequent and annoying mistake.

To display semaphore lights, we wrote separate methods for the train and for the truck. They calculate some essential points and display a colored rectangle along the track on each side of the crossing:

displayTrainSemaphoresWithColor: aColorValue

"Display train semaphore lights as rectangles of specified color."

```
| limit rectangle |
limit := self trainCrossingLimit.
rectangle := 0 @ 0 corner: 4 @ 4.
self graphicsContext
  paint: aColorValue;
  displayRectangle: rectangle at: self crossing + (limit negated @ 3);
  displayRectangle: rectangle at: self crossing + (limit - 4 @ -6)
```

The application now works and it only remains to improve the implementation of a few awkward methods. As an example, the last method could be improved by defining a `lightRectangle` variable and calculating the coordinates from the rectangle instead of using fixed numeric values. We could then change the size of the lights without having to change the calculation of the position of the rectangle. We leave this as an exercise. A non-trivial improvement of the user interface is the subject of the next section.

Main lessons learned:

- Class Semaphore makes it possible to synchronize processes.
- The essence of the operation of a Semaphore are messages signal and wait.
- If a Process sends wait to a Semaphore and the difference between the number of signal and wait messages received by the Semaphore is not at least 1, the Process is suspended until the condition is satisfied.
- A deadlock occurs when several processes wait for one another and cannot proceed.
- Processor yield moves the currently active process to the end of its queue of runnable processes.

Exercises

1. Complete and clean up the implementation as suggested. One of the questions to resolve is whether the model instance variable of Vehicle implies that TrainSimulation should be a Model of Vehicle and if so, how this part of the application should be reimplemented.
2. Replace train and truck rectangles with arrows and reverse their direction at the end of the track.
3. Modify Train and Truck so that they know how to display themselves.
4. Change the simulation program so that the train and the truck each have their own speed controlled by its own slider.
5. Implement vehicle redrawing using VisualComponent's animation protocol. In our case, the code is not simpler but its implementation is an interesting introduction to animation.
6. In our design, the train and the truck can never collide or become deadlocked. Real life crossing are not, unfortunately, so perfect because a driver can overlook a traffic light and a traffic light may malfunction. Modify our design so that the user can set a <0,1> probability that a failure occurs.

7. Add an animation of a crash of the two vehicles.
8. Extend the simulation program to several parallel train and truck tracks with a single shared crossing. Each track has its own vehicle running at its own speed and each vehicle's speed is controlled independently. Speed is controlled by a single slider and a set of radio buttons, one for each vehicle.
9. Simulation of events occurring in parallel, such as our train simulation, are common. Define a simulation framework in which such simulations could be implemented as easily as possible and test it on our example.
10. Does process priority have any effect on its behavior with respect to Semaphore?
11. What would happen if we removed a Semaphore with a queue of waiting processes?
12. What happens when you close the simulation window without terminating the two vehicle processes?
13. Study class Semaphore and provide detailed answers to the following questions:
 - a. What happens to a process when it sends wait and the Semaphore is ready to let it run?
 - b. What happens if the Semaphore is not ready to let the process run?
 - c. What happens to the process that sends signal to a Semaphore?
 - d. What happens when several wait messages are sent to the same Semaphore?
 - e. What happens when several signal messages are sent to the same Semaphore?

13.4 Making train simulation layout customizable

We now have a working train simulation but we would like to make it more flexible by allowing the user to customize it. In particular, we would like to make it possible to stop simulation and grab the line representing a track and move it to a new position without changing its length, and grab an endpoint of a track and move it to change the track's length.

In both cases, horizontal lines will remain horizontal and vertical lines will remain vertical. Both actions will use a special cursor displayed only when the mouse moves over a track or a track endpoint (Figure 13.12).

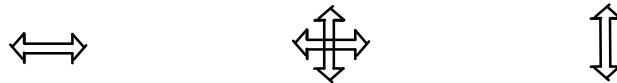


Figure 13.12. Special cursor shapes for dragging tracks and changing their length.

The details of the desired behaviors are described in the following scenarios:

Scenario 1: User moves the cursor over the layout subview

1. *User* moves cursor into the hot area around the right end of the horizontal track.
2. *Program* changes cursor to the shape shown on the left in Figure 13.12.
3. *User* continues moving the cursor until it leaves the hot area.
4. *Program* changes cursor to normal shape.
5. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track.
6. *Program* changes cursor to the shape shown in the center of Figure 13.12.
7. *User* moves mouse cursor into the hot area surrounding the inner portion of the vertical track.
8. *Program* maintains cursor shape unchanged.
9. *User* moves mouse cursor into the hot area around the upper end of the vertical track.
10. *Program* changes cursor to the shape shown on the right of Figure 13.12.

Scenario 2: User drags the horizontal track to a new position

1. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track.
2. *Program* changes cursor to the shape shown in the center of Figure 13.12.
3. *User* presses the <operate> button and moves the mouse while holding the button down.
4. *Program* drags the track maintaining its length and horizontal orientation, and follows the cursor without changing its shape. The program also moves semaphore lights and the vehicle and enforces constraints such as keeping the track within the subview.

5. *User* releases the mouse button.
6. *Program* stops dragging the track.

Scenario 3: User drags the left end of the horizontal track to a new position

1. *User* moves the cursor into the hot area surrounding the left end of the horizontal track.
2. *Program* changes cursor to the shape shown on the left of Figure 13.12.
3. *User* presses the <operate> button and moves the mouse while holding the button down.
4. *Program* drags the end of the track, restricting motion to horizontal displacement and following the cursor without changing its shape. The program also moves the vehicle if necessary and enforces constraints such as limiting the track to the subview. The rest of the track is not affected.
5. *User* releases the mouse button.
6. *Program* stops dragging the endpoint of the track.

The scenarios show that the details are not as trivial as we may have thought - and this is one of the reasons why scenarios are useful. Among other things, the definitive solution will require a specification of the details of the constraints on track motion. We suspect that the details will not be trivial and we will thus ignore them in the first iteration (leaving them as an exercise:), concentrating on unconstrained motion and leaving the details for another iteration.

Preliminary Design

The major modification of the existing implementation is that we must replace the current *NoController* with a controller that can handle the desired interactions. We will call this controller *LayoutController* and its essential responsibilities are as follows:

1. Control the shape of the cursor.
2. As the mouse moves in the dragging mode of operation, keep redrawing the appropriate part of the layout without affecting the rest. This task consist of
 - a. continuously redrawing a *track*,
 - b. redrawing the *vehicle* on the track being moved,
 - c. redrawing *semaphore lights* of both tracks, and
 - d. updating instance variables describing the tracks.

To get a better understanding of the details, let's refine our scenarios into class level conversations.

Scenario 1: User moves the cursor over the layout subview

1. *User* moves cursor into the hot area around the right end of the horizontal track.
2. *LayoutController* tracks mouse position and changes cursor to the appropriate shape.
3. *User* continues moving the cursor until it leaves the hot area.
4. *LayoutController* tracks mouse position and changes cursor to normal shape.
5. etc.

Scenario 2: User drags the horizontal track to a new position

1. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track
2. *LayoutController* tracks mouse position and changes cursor to the appropriate shape.
3. *User* presses the <operate> button and moves the mouse around while holding the red button down.
4. *LayoutController* detects mouse button press and starts converting mouse movement events into the following actions:
 - a. Calculate mouse displacement with respect to the previous position.
 - b. Obtain current track position and other relevant information from *TrainSimulation* and combine it with mouse displacement to calculate new parameters.
 - c. Send new layout parameters to *TrainSimulation*.
 - d. Calculate rectangles that must be redrawn and ask *LayoutView* to invalidate them immediately.
5. *User* releases the mouse button.
6. *LayoutController* senses mouse button release and changes its mode of operation to 'no dragging.'

Scenario 3: User drags the left end of the horizontal track to a new position

Very similar to Scenario 2; only the details of parameters and calculations are different.

The preliminary Object Diagram updated from previous section is shown in Figure 13.13.

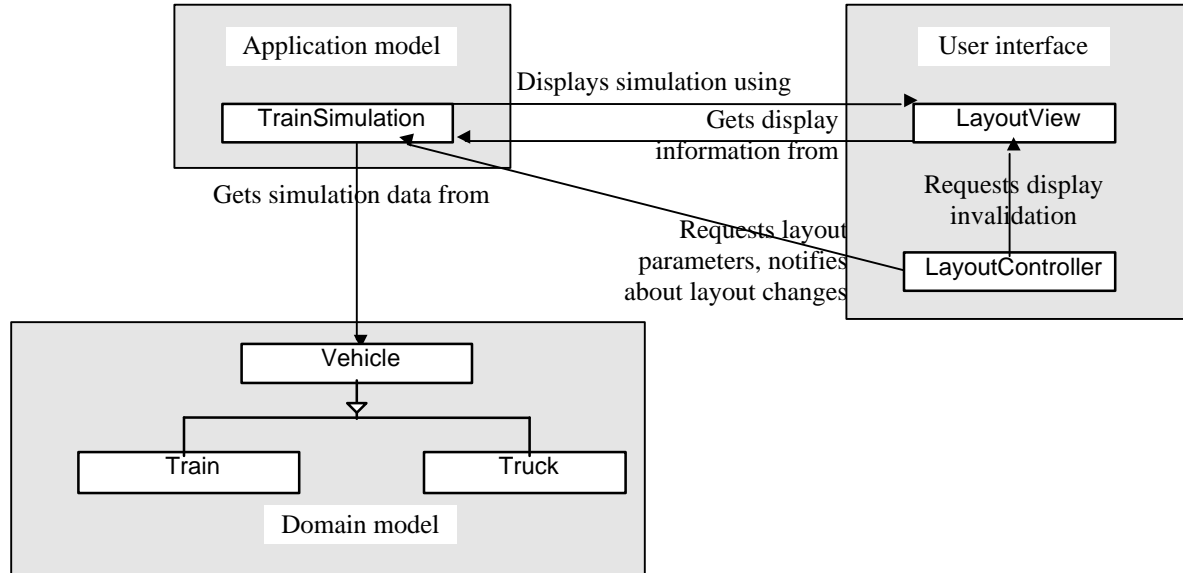


Figure 13.13. Preliminary Object Model of customizable train simulation.

Design Refinement

Class Descriptions

The existing classes don't acquire any new responsibilities but this does not necessarily mean that we don't have to change them. In our existing implementation, we cached layout parameters in the view and initialized them once and for all; this would not work any more. Those cached layout parameters that affect the display and the simulation now have to be accessed via accessing methods, and the code that uses them must change accordingly. We are getting a lesson in the usefulness of using accessing methods rather than accessing variables directly.

Since the essential responsibilities of existing classes don't change, we will not rewrite their descriptions and limit ourselves to the controller class.

LayoutController. Responds to mouse events by displaying appropriate cursor shapes and performing drag operations on tracks and their end points. Does not have an <operate> menu.

Superclass: Controller

Components: endArea (Rectangle) holds the shape of the hot area for end points, isDragging (Boolean) keeps track of current mode of operation, crossArrowsCursor (Cursor), horizontalArrowsCursor (Cursor), verticalArrowsCursor (Cursor), tracking (Symbol such as #horizontalTrack or #topVerticalEnd) - keeps track of the kind of area under the cursor, newCursorPosition (Point) - current position, oldCursorPosition(Point) - cursor position before last mouse move event

Behaviors

- initialization - initialize cursors and define end point area
- event handling - intercept cursor movement and red button events
 - cursor display - display area-dependent shape
 - dragging - drag tracks or their endpoints in response to mouse move event

Collaborators

LayoutView

- accessing – obtain layout parameters from domain model

TrainSimulation

The Hierarchy Diagram updated from the previous section is shown in Figure 13.14; the Object Model does not change.

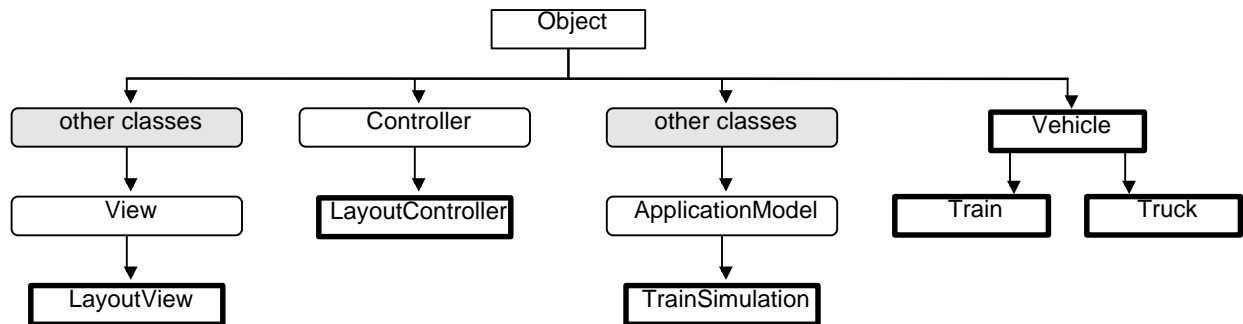


Figure 13.14. Class Hierarchy Diagram with classes to be designed shown as heavy rectangles.

Implementation

We will now show some of the code of `LayoutController`. Note again that this is a first iteration and that we are leaving many details such as dragging constraints and optimized invalidation for the next iteration.

Class `LayoutController`

Initialization

initialize

"Initialize mode of operation, cursors, and hot area rectangle for track end points."

isDragging := false.

horizontalArrowsCursor := Cursor

image: self class horizontalArrows asImage

mask: self class horizontalArrowsMask asImage

hotSpot: 8 @ 8

name: 'horizontal arrows'.

verticalArrowsCursor := Cursor

image: self class verticalArrows asImage

mask: self class verticalArrowsMask asImage

hotSpot: 8 @ 8

name: 'horizontal arrows'.

crossArrowsCursor := Cursor

image: self class crossArrows asImage

mask: self class crossArrowsMask asImage

hotSpot: 8 @ 8

name: 'horizontal arrows'.

endArea := -4 @ -3 corner: 4 @ 3

As in the Chess program, we used Image Editor to create cursors and their masks.

Mouse events

These methods redefine some of the events protocol inherited from class `Controller`.

mouseMovedEvent: event

"The mouse has moved. Respond according to current mode of operation."


```
| areaCursor |
isDragging
  ifTrue: "Perform method corresponding to parameter being tracked."
    [newCursorPosition := self sensor cursorPointFor: event.
     self perform: tracking.      "Execute message appropriate for parameter."
     oldCursorPosition := newCursorPosition]
  ifFalse: "We could have entered a new area. Check and update cursor."
    [areaCursor := self hotSpotAreaCursorFor: (self sensor cursorPointFor: event).
     Cursor currentCursor: areaCursor]
```

The auxiliary method `hotSpotAreaCursorFor:` selects cursor shape according to cursor location:

hotSpotAreaCursorFor: aPoint

"Determine which parameter is being tracked from the location of the cursor. Return appropriate cursor."

```
self getTrackingParameterFor: aPoint.
^self selectCursor
```

where

getTrackingParameterFor: aPoint

"Return symbol representing tracking parameter."

```
(self bottomVerticalTrackAreaEndpoint containsPoint: aPoint)
  ifTrue: [^tracking := #bottomVerticalEnd].
(self topVerticalTrackAreaEndpoint containsPoint: aPoint)
  ifTrue: [^tracking := #topVerticalEnd].
(self verticalTrackArea containsPoint: aPoint)
  ifTrue: [^tracking := #verticalTrack].
(self leftHorizontalTrackAreaEndpoint containsPoint: aPoint)
  ifTrue: [^tracking := #leftHorizontalEnd].
(self rightHorizontalTrackAreaEndpoint containsPoint: aPoint)
  ifTrue: [^tracking := #rightHorizontalEnd].
(self horizontalTrackArea containsPoint: aPoint)
  ifTrue: [^tracking := #horizontalTrack].
^tracking := nil
```

determines which parameter we are tracking, and

selectCursor

```
(#(#leftHorizontalEnd #rightHorizontalEnd) includes: tracking)
  ifTrue: [^horizontalArrowsCursor].
(#(#bottomVerticalEnd #topVerticalEnd) includes: tracking)
  ifTrue: [^verticalArrowsCursor].
(#(#horizontalTrack #verticalTrack) includes: tracking)
  ifTrue: [^crossArrowsCursor].
^Cursor normal
```

finds the appropriate cursor. We think that the two last methods could be implemented more elegantly and leave this task as an exercise. Individual hot areas are returned by auxiliary methods such as

bottomVerticalTrackAreaEndpoint

"Calculate hot area for the bottom of the current vertical track by shifting the pre-initialized endArea rectangle by a value equal to the end point of the vertical track."

```
^endArea translatedBy: (self truckTrack at: 2)
```

where `truckTrack` is an array of track end points calculated dynamically from current track coordinates as follows:

truckTrack

"Return top and bottom end points of vertical track."

```
| start |
```

```
^Array with: (start := model truckTrackStart)
with: start + (0 @ model truckTrackLength)
```

Returning now to red button mouse events, the following method is responsible for starting dragging:

redButtonPressedEvent: event

"User pressed red button. If we are 'tracking' (cursor in a hot area), change mode to 'dragging' and initialize oldCursorPosition for further updates of layout during dragging. The rest is done by mouseMovedEvent:."

```
tracking isNil "Are we 'tracking' a parameter - is cursor in a hot area?"
```

```
ifFalse: [isDragging := true.
```

```
oldCursorPosition := self sensor cursorPointFor: event]
```

and once this method is executed, dragging occurs via the mouseMovedEvent: defined above. When the user releases the button, the following event method returns operation to the non-dragging mode:

redButtonReleasedEvent: event

"User released red button, switch mode to non-dragging."

```
isDragging := false
```

Dragging

Dragging operations are performed only in the dragging mode (see mouseMovedEvent: above) and are based on invalidation. As an example, dragging of the entire horizontal track is implemented as follows:

horizontalTrack

"User moved the cursor in the dragging mode. Drag the horizontal track to reflect the new cursor position."

```
| shift track trackStart trackLength |
```

```
"Update affected track parameters in the model - TrainSimulation."
```

```
shift := newCursorPosition - oldCursorPosition.
```

```
trackStart := model trainTrackStart + shift.
```

```
model trainTrackStart: trackStart.
```

```
trackLength := model trainTrackLength.
```

```
"Calculate the end points of the track for the view."
```

```
track := Array with: trackStart with: trackStart + (trackLength @ 0).
```

```
"Ask LayoutView to invalidate the area around the track immediately."
```

```
view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)
```

```
repairNow: true
```

We use horizontalTrackInvalidationAreaForTrack: to calculate the rectangle to be invalidated by LayoutView. Since the current iteration is just a proof of concept, we define the area simply as the rectangle given by the old and the new cursor position, the track, and 30 pixels around on all sides. This area includes all semaphore lights, the vehicle, and the text. It is, of course, unnecessarily large since we only need to redraw the track itself, the vehicle, the lights, and the text, and the combination of these rectangles is only a fraction of the rectangle that we are invalidating. We leave an optimized implementation as an exercise.

horizontalTrackInvalidationAreaForTrack: anArray

"This is the area to redraw. It is too generous but it works. Optimization is left until next iteration."

```
^Rectangle origin: (anArray at: 1) + (-30 @ -30)
```

```
corner: (anArray at: 2) + (30 @ 30)
```

As another example of dragging, the left end of the horizontal track is dragged as follows:

leftHorizontalEnd

"Drag the left end point of the horizontal track. Position of vehicle is not changed because it is relative to the start of the track."

```
| shift trackStart trackLength track |
```

```
"Recalculate affected TrainSimulation parameters."
```

```
shift := newCursorPosition x - oldCursorPosition x. "Movement is restricted to horizontal."
```

```
trackStart := model trainTrackStart + (shift @ 0).
```

```
trackLength := model trainTrackLength - shift.  
model trainTrackStart: trackStart.  
model trainTrackLength: trackLength.  
"Calculate both end points of track."  
track := Array with: trackStart with: trackStart + (trackLength @ 0).  
"Ask view to redraw the affected part."  
view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)  
    repairNow: true
```

The method for dragging the right end is slightly simpler because it affects only the length and not the start of the track:

rightHorizontalEnd

" Drag the right end point of the horizontal track. Position of vehicle is not changed because it is relative to the start of the track."

```
| shift trackStart trackLength track |  
"Recalculate affected TrainSimulation parameters."  
shift := newCursorPosition x - oldCursorPosition x.    "Movement is restricted to horizontal."  
trackStart := model trainTrackStart.  
trackLength := model trainTrackLength + shift.  
model trainTrackLength: trackLength.  
"Calculate both end points of track."  
track := Array with: trackStart with: trackStart + (trackLength @ 0).  
"Ask view to redraw the affected part."  
view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)  
    repairNow: true
```

The two methods for dragging track end points are almost identical and it would be better if we factored out the shared behavior. This would make the code more readable and guarantee that if we make changes, they will affect both methods identically. This stylistic improvement is left for the next iteration.

The remaining dragging methods follow the same pattern and the only other required code is simple accessing methods.

Main lessons learned:

- To simulate dragging of geometric objects, monitor the mouse-moved event and repeatedly erase the object in its original position and redraw it in the new position. Use `invalidateRectangle: aRectangle repairNow: true` to obtain clean response..An alternative approach is to use the animation protocol in class `VisualComponent`.
- Using accessing methods instead of accessing instance variable directly makes code easier to modify.

Exercises

1. Test and explain what happens if we don't specify immediate invalidation.
2. Complete and test the implementation.
3. Optimize the calculation of the invalidation area and implement constraints on moving a track and its end points. Note that if the user moves the cursor fast and the invalidation rectangle is not sufficiently large, invalidation may not be satisfactory.
4. Extend the controller to allow the user to grab a crossing light and move it to a new position. Its corresponding other light should move accordingly.
5. Extend the controller to allow the user to reposition a vehicle on its track. Note that the operation of the crossing imposes constraints on allowed vehicle configurations.

Conclusion

Computer operation generally requires controlled ‘simultaneous’ operation of more than one program. Such parallel threads of execution are called processes. Our definition of the concept of a process is that a process is a program that shares the CPU and other resources with other programs under the control of a process scheduler. VisualWorks processes use pre-emptive process scheduling which leaves a process to run until it terminates or suspends its execution or is suspended by a higher priority process or a semaphore. VisualWorks processes can be divided into two broad categories - system processes and user processes. Applications written by users and code segments executed from the workspace are normally executed as user processes and system processes include tasks such as garbage collection and input/output tracking..

VisualWorks processes are instances of class **Process**. Processes are always derived from blocks and the creation protocol can be divided into methods that create a process and schedule it for execution immediately, and methods that create a process but don’t schedule it for execution. The first ones are defined in **BlockClosure**, the second in class **Process**.

Processes can be active, suspended, or terminated and the methods that change process state are **suspend**, **resume**, and **terminate**. Message **terminate** shuts down and destroys the process, **suspend** takes the process out of the queue of runnable processes but does not destroy it, and **resume** puts a suspended process at the end of the queue of runnable processes.

Every Smalltalk process has a priority expressed as an integer number between 1 (the lowest) and 100. User processes run by default at priority 50, I/O processes such as mouse and keyboard events run at priority 90, real-time processes run at priority 100, and system background processes run at priority 10. The **fork** creation message creates a **Process** at the priority of the parent process which created it, message **forkAt: priority** creates a **Process** at the specified priority. Both **fork** messages immediately schedule the new process which means that it is placed at the end of the queue of runnable processes of their priority.

Smalltalk processes are managed by **Processor**, the single instance of class **ProcessScheduler**. **Processor** always selects for execution the first process in the highest priority non-empty queue of runnable processes. Since **Processor** controls all priority queues, its protocol provides a very useful complement to other process control messages. Sending **yield** to **Processor** moves the currently active process to the end of its queue and activates the process at the head of the queue.

Processes often need to coordinate their operation in a way similar to traffic control at an intersection. Smalltalk’s process coordination is based on analogy with traffic lights implemented by class **Semaphore**. The essence of the operation of a **Semaphore** are messages **signal** and **wait**. A **signal** is a notification by a **Process** to a **Semaphore** that it no longer needs its protection and corresponds roughly to turning a traffic light green.

When a process wants to gain access to a resource protected by a **Semaphore**, it sends the semaphore the **wait** message which is equivalent to attempting to proceed through an intersection and to turn the traffic light of the other street red. If the difference between **signal** and **wait** messages previously sent to the **Semaphore** is not positive, the sending **Process** is suspended until another process sends **signal**.

When several processes run, a deadlock can occur making it impossible for any of the processes to continue. As an example, a deadlock occurs when process X is suspended, and process Y waits for X to send **signal** to its **Semaphore**.

In the last section of this chapter, we extended our process-oriented example of train simulation with interactive control of the graphical layout. To simulate dragging of geometric objects, we monitored the mouse-moved event and repeatedly erased the object from its original position and redrew it in the new position. We used **invalidateRectangle: aRectangle repairNow: true** to obtain fast response without missed display operations. The same principle (erasing and redrawing) is the basis of all computer animations.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Process – represents a thread of execution that monopolizes the CPU and other computer resources.

ProcessScheduler - class whose single instance **Processor** is in charge of scheduling processes. The basis of the whole operation of Smalltalk.

Semaphore - a process-synchronizing class used to restrict access to a shared resource. An analog of traffic lights.

Widgets introduced in this chapter

slider - interactive graphical representation of a floating point number model.

Terms introduced in this chapter

process - independent thread of execution that takes turns using the CPU and other computer resources with other processes; created from and executing a block

process scheduler - object controlling the order in which processes are activated

process priority - a number between 1 and 100 which determines relative priority of a process; the highest priority runnable processes are executed first; 1 is the lowest process priority

runnable process - a process ready to be executed and waiting for its turn

semaphore - an object controlling activation of processes sharing a common resource; computer equivalent of traffic lights

scheduling - controlling the order in which processes execute

signal - message to a semaphore equivalent to changing a traffic light to green

wait - message to a semaphore equivalent to attempting to pass an intersection and set the traffic light of the other street to red; can only be satisfied if the semaphore is currently "green"

suspending a process - stopping the execution of a process and taking it out of the queue of runnable processes without destroying it; can be reversed by resume

resuming a process - returning a suspended process to its priority queue of runnable processes

terminating a process - terminating process execution and destroying it

yielding a process - sending the currently active process to the end of its priority queue

Abstract class. A class that is never instantiated. Used to factor out properties shared by its subclasses. As an example, class **Boolean** is an abstract class with two concrete subclasses **True** and **False**.

Abstract data type.

Abstraction.

Accelerator.

Action button. Component of *user interface* looking like a button. Its activation by the mouse in Smalltalk sends a message.

Algorithm. An orderly sequence of steps describing the solution of a problem such as sorting a collection of names alphabetically or finding the largest of a set of numbers.

Application.

Application model. A class implementing the connection between the user interface (buttons, checkboxes, input fields, sliders, etc.) and the domain model consisting of objects that represent components of the problem domain. The user interface sends messages to the application model, which converts them to messages to specific objects in the domain model. The communication also works that way in the opposite direction. Example: *User interface* - buttons, sliders, and other widgets on the screen triggering and displaying deposits, withdrawals, account balances; *domain model* - objects representing accounts, transactions, and bank clients; application model - the object converting signals from the interface to messages to domain objects and vice versa.

Argument. An object sent as a parameter of a message. As an example, in message **3 + 17** object **3** is the receiver and object **17** is the argument. Message **5 between: 2 and: 17** has two arguments, objects **2** and **17**.

ASCII. American Standard Code for Information Interchange. An internationally used code for representing printable and control characters as one-byte code. The most widely used representation of textual data.

ASCII file. File whose contents are to be interpreted as ASCII codes and thus displayable on a screen as text. Compare with *binary file*.

Association. A pair consisting of a 'keyword' and its 'value' as in a dictionary. The basic component of a *dictionary* structure.

Binary file. File whose contents are not to be interpreted as ASCII codes but rather as encoded data or instruction codes of an executable program. Consequently not displayable on a screen as text unless interpreted by a special program. Compare with *ASCII file*.

Binary message. A message composed of one or two special characters including **,+-@/** with a single argument as in **3 + 4** or in **3 @ 17**.

Binding

Block. Short for *Block closure*.

Block argument. Variable declared inside a block as in **[:index | array at: index]**. Its scope is restricted to the block.

Block closure. One or more statements possibly preceded by block arguments, surrounded by square brackets as in **[x sin]**, **[x := rad sin. y := rad cos]**, **[x : y | Transcript show: x, y]**.

BOSS.

Browser. A multi-window Smalltalk tool providing access to source code of all declarations in the library. Used to examine existing code and to create and edit new code. Several browsers are provided ranging from a system wide system browser to browsers restricted to a specific class or method.

Canvas.

Cascading. A shorthand allowing sending a sequence of messages to the same receiver without repeating the receiver as in **Transcript cr; show: 'abc'; cr** instead of **Transcript cr. Transcript show: 'abc'. Transcript cr**.

Category. Grouping of related classes enforced by the *browser*. Helps to reduce the multitude of classes in the library into a more manageable set of categories, makes it possible to *file out* and *file in* a group of classes by a single command. Does not have any semantic meaning and moving a class from one category to another thus has no effect on its operation.

Chaining. Using the result of a statement as an argument or as the receiver of a message as in **Transcript show: (3 factorial) printString** where **3 factorial** is the receiver of **printString**, and **3 factorial printString**

is an argument of show: Makes code more compact, may save temporary variables, but makes code more difficult to read and should not be overused.

Changes file.

Character. Usually refers to a symbol that can be displayed on the screen although some characters have only 'control' functions such as starting a new line.

Check box.

Class. A mold from which objects are created - a formal specification of an abstract template of which individual objects are instances. Contains the name of the class and its superclass, its variables, and methods. All instances of a class (objects created from its specification) share all variables and can execute all methods declared in the class and its superclasses.

Class instance variable. A variable declared with the 'class' button of the browser active, inherited by all subclasses, and accessible to all of their class and instance methods. Unlike a *class variable*, a class instance variable is distinct for each subclass that inherits it. Unlike class methods which are accessible to classes but not their instances, all types of class variables are accessible both to classes in which they are declared and their instances.

Class method. A method whose receiver must be the class in which the method is declared or its subclass. Compare with *instance method*. Used mainly for creation of class instances (as in `Array new` or `Array new: 3`), for initialization of class variables (as in `Random initialize`), and for generally useful tasks that do not require the creation of an instance (as in `Dialog warn: 'This operation is illegal'`).

Class variable. A variable declared with the 'class' button of the browser active, inherited by all subclasses, and accessible to all of their class and instance methods. Unlike a *class instance variable*, a class instance variable is shared by the class in which it is declared and its subclasses; consequently, a change of its value by one class affects all other classes that share it. Unlike class methods which are accessible to classes but not their instances, all types of class variables are accessible both to classes in which they are declared and their instances.

Clean block. A block that does not depend on the context in which it is used. As an example, `[Dialog warn: 'Illegal operation']` is clean while `[Dialog warn: warningMessage]` is not because it depends on the value of variable `warning` block declared and manipulated outside the block.

Collection. *Abstract class* holding all properties shared by concrete collections such as `Array`, `OrderedCollection`, and `Dictionary`.

Compilation. The process of converting the human readable source code to internal representation executable by the computer. In Smalltalk compilation occurs, for example, upon activation of the 'accept' command in the browser or the debugger or upon activation of 'do it' in the workspace. In the latter case, compilation is immediately followed by execution of the code, in the former case, the compiled code is only internally stored.

Concatenation.

Concrete class. A class declared with the intent to be used to create instances. As an example, class `Character` is a concrete class whose instances are widely used, mainly as elements of strings. See also *Abstract class*.

Control statement. Usually a composite statement that controls how its components are executed. Examples: `ifTrue:`, `whileFalse:`, `do:` which select how many times their components are executed or which of their components are executed.

Controller. In the Smalltalk *MVC* (model-view-controller) paradigm, that part of the user interface which monitors users input (keyboard and mouse) and communicates it to the model which then decides whether the action should cause a change in the displayed view, and notifies the view to change itself if necessary. Each active widget (button, input field, etc.) has a controller monitoring and processing its activation, a model to which it communicates all activity, and a view which takes care of redisplaying itself if requested to do so by the model.

Data abstraction.

Data structure. An established view and organization of data with attached access mechanisms such as an array, a set, or a dictionary. A composite object containing several components is also a form of a data structure.

Dataset widget. Similar to a table but used to display rows of objects with identical structure consisting of multiple components, such as rows of course marks for individual students.

Decompilation. The reverse of *compilation*. Converts compiled code to *source code*. Since compiled code does not contain information about variable names, decompiled code contains artificially created variable names instead of the original ones. Decompiled code may differ from source code because the compiler performs certain optimizations to improve the efficiency of execution.

Debugger. A multi-window Smalltalk tool allowing step-by-step execution, inspection of variables, and modification of source code and variable values. Used for debugging new code and for examining operation of existing code.

Debugging. Step-by-step execution performed to locate and correct mistakes (bugs) in a program.

Declaration. Formal specification of the name of a class and its parameters, its comment, or its methods. Must satisfy the syntactic and semantic rules of Smalltalk and is performed using a browser.

Default. Value to which an object is initialized when it is created. Normally nil, unless the initialization method explicitly initializes the object to a different value. As an example, buttons are initialized to display themselves with black text on gray background.

Dependency. A mechanism built into Smalltalk that allows mutually dependent objects to notify one another automatically when they change. Used by the *MVC paradigm*: When the *model* changes, it automatically notifies all its dependent *views* which then change their appearance if the change of the model requires it.

Dictionary. A *set of associations* - a collection of associations similar to a real life dictionary (but not sorted).

Domain model. That part of an application that represents the problem domain objects such as bank accounts, clients, deposits, withdrawals, and etc., in a banking application. The usually graphical appearance that the application presents to the user in the form of buttons, labels, etc. (the *user interface*), is connected to the domain model through the *application model*. Note that an application could represent the same domain model using several different user interfaces each requiring its own application model. Similarly, a given user interface could be used with several different domain models, each again requiring its own application model. In a narrower sense, each class representing a part of a problem domain, for example a class representing a bank account, is also called a domain model.

Double dispatching. Executing a message by sending it to the argument object, with the original receiver being used as argument. Used in arithmetic messages to simplify decision making and to make declaration of arithmetic messages more general. As an example, execution of $3 + 5.4$ (message `+` with an integer receiver and a float argument) consists of sending message `5.4 sumFromInteger: 3` (message `sumFromInteger:` with a float receiver and an integer argument). Whereas the first form would require determining the type of the argument and executing appropriate form of addition, the second form does not require any tests because method `sumFromInteger:` is declared to deal with integer arguments.

Distinguished instance. Instance of a class that is intended to be unique. As an example, classes `True`, `False`, and `UndefinedObject` each have a single instance.

Dynamic binding. Languages using dynamic binding, such as Smalltalk, do not require the specification of the class to which an identifier denoting a variable or an argument must belong. This binding is performed when an object is assigned to the identifier during program execution. Main advantage: code may be very general because it may be executable with instances of many different classes. Disadvantage: Does not provide the extra degree of security provided in languages with *static typing* where the compiler can recognize improper use of variables even before the code is executed.

Editor widget. A widget that can display text on multiple lines and automatically adjust the contents of the lines to its current size. A writeable editor widget allows the user to cut, paste, edit, and perform similar operations, in the case of a code view also to execute code. When the execution ability is missing, the widget is usually referred to as a text editor.

Encapsulation. Bundling of the information held by an object and of its functionality into a single entity. Compare to procedural languages in which data is separated from functionality.

Expression.

File. Binary representation of data or code or a combination of both stored on a secondary storage device such as a disk or a CD ROM, under a user selected file name.

File in. The procedure of reading and automatically compiling and inserting into the library of a file containing the code of a method, a protocol, a class, or a category. The file must be in a special format produced by the *file out* command. The 'file in' command is available in file browsing tools.

File out. Command available in file browsing tools and used to store a method, a protocol, a class, or a category in a file, using a special form expected by the *file in* command. Used for transporting source code from one computer to another. The command is available in various Smalltalk browsers.

Finalization. Actions executed by the program when a window is being closed.

Floating point number. Internal computer representation of numbers with fractional components such as 3.14 or -0.00143. Compare to *integer*. In Smalltalk, represented by several different classes depending on magnitude and number of significant digits.

Framework.

Global variable. A variable accessible to any object and at any time, automatically saved when the Smalltalk environment is saved and automatically restored when Smalltalk is started up. Global variables are stored in the Smalltalk dictionary along with all other globally accessible objects including classes and pool dictionaries. This makes global variables dangerous because naming a global variable with the name of an existing class replaces the class with the global variable and destroys it.

Graphical user interface (GUI) - the collection of windows, buttons, labels, and other widgets seen by the user of a computer application

Icon. Visual representation of a collapsed window. In Smalltalk, the user can assign any image to the icon of a window.

Image file.

Immediate object. Object whose internal representation is not accessed by a pointer. In fact, the 'pointer' is the value of the object. Includes *SmallInteger* and *Character*.

Indexable variable. An instance variable or an object whose elements are accessed by index, as in an array.

Information hiding. The rule that internal variables of an object can only be accessed through methods explicitly declared for this purpose in the object's class. As an example, a point's x and y coordinates can be obtained and changed only because class *Point* contain declaration of messages to do this. Without these methods, the values of x and y would not be accessible.

Inheritance. The mechanism through which all variables and methods declared in a class are valid for all its subclasses as well. As an example, class *Collection* declares method *contents* and all its subclasses (*Array*, *Set*, *Dictionary*, etc.) thus also understand this message. Similarly, class *ControllerWithMenu* declares variable *menuHolder* and all its subclasses thus automatically inherit it too.

Input field widget. A one line text widget with editing capabilities such as cut, copy, and paste. May be writeable or read only.

Inspector. A Smalltalk tool opening a two-view window listing the components of the object being inspected in one subview, and the value of the selected coonent in the adjacent view.

Instance. Synonym of *object*. A realization of a class with specific values for each instance variable declared or inherited by the defining class. As an example, 3 and 5.1 are instances of class *SmallInteger* while individual labeled action buttons on a window are instances of class *ActionButton*.

Instance method. A method declared with the 'instance' button of the browser on. Instance methods can only be sent to instances of a class. Compare with *class methods* which can only be sent to the class itself but are not understood by instances.

Instance variable. Variable holding an object defining one of the parameters of an *instance* of a class. Each instance of the same class has the same instance variables but each may have a different value. Compare with *class variable* and *class instance variable*.

Integer. Internal representation of the mathematical concept of an integer, a number without a fractional point. In Smalltalk implemented by several classes for different ranges of magnitude.

Interval.

Iteration.

Keyword

Keyword message

Label widget.

Launcher.

List widget.

Literal.

Loop.

Menu bar widget.

Menu button.

Message.
Metaclass.
Method.
Model.
Mouse buttons.
Multiple inheritance.
MVC paradigm.
Object.
Object file.
Optimization.
Order of evaluation.
Ordered collection.
Overloading.
Palette.
Pane.
Parameter.
Persistent object.
Pointer.
Pool dictionary.
Pop-up menu.
Polymorphism.
Private method.
Primitive method.
Private method.
Protocol.
Pseudovisible.
Radio button.
Receiver.
Return object.
Reusability.
Selector.
self.
Semantics.
Set.
Shortcut key.
Signature.
Simulation.
Single inheritance.
Slider widget.
Source code.
Standard protocol names.
State.
State variable.
Statement.
Static binding.
Stream.
String.
Symbol.
Syntax.
Subclass.
Superclass.
super.
Temporary variable.
Text.
Text editor widget.

Text file.
Transcript.
UI.
User interface.
Unary message.
Undeclared variable.
Value holder.
Variable.
View.
Widget.
Wildcard.
Window.
Workspace.

References

Books

Kent Beck: Smalltalk: Best Practice Patterns, Prentice-Hall, 1997.
Timothy Budd: A Little Smalltalk, Addison-Wesley, 1987.
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
Adele Goldberg: Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1983.
Adele Goldberg, David Robson: Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1985.
Adele Goldberg, David Robson: Smalltalk-80, The Language, Addison-Wesley, 1989.
Trevor Hopkins, Bernard Horan: Smalltalk: An Introduction to Application Development Using VisualWorks, Prentice-Hall, 1995.
Tim Howard: The Smalltalk Developer's Guide to Visual Works, SIGS Books, 1995.
Ted Kaehler, Dave Patterson: A Taste of Smalltalk, Norton, 1986.
Wilf Lalonde: Discovering Smalltalk, Addison-Wesley, 1994.
Wilf Lalonde, John R. Pugh: Inside Smalltalk, Volume I, II, Prentice-Hall, 1991.
Wilf Lalonde, John R. Pugh: Smalltalk/V, Practice and Experience, Prentice-Hall, 1994.
Simon Lewis: The Art and Science of Smalltalk, Prentice-Hall, 1995.
Chamond Liu: Smalltalk, Objects, and Design, Prentice-Hall, 1996.
Mark Lorenz: Rapid Software Development with Smalltalk, SIGS Books, 1995.
Lewis J. Pinson: An Introduction to Object-Oriented Programming and Smalltalk, Addison-Wesley, 1988.
Jonathan Pletzke: Advanced Smalltalk, Wiley, 1997.
Wolfgang Pree: Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1994.
Timothy Ryan: Distributed Object Technology, Concepts and Applications, Prentice-Hall, 1996.
Dusko Savic: Object-Oriented Programming with Smalltalk, Prentice-Hall, 1990.
Susan Skublics, Edward J. Klimas, David A. Thomas: Smalltalk with Style, Prentice-Hall, 1996.
Dan Shafer, Dean A. Ritz: Practical Smalltalk, Springer-Verlag, 1991.
Dan Shafer, Scott Herndon: IBM Smalltalk Programming for Windows and OS/2, Prima, 1995.
David N. Smith: IBM Smalltalk: The Language, Benjamin Cummings, 1995.

Periodicals

JOOP – Journal of Object-Oriented Programming, SIGS Publications
Object Magazine, SIGS Publications
Smalltalk Report, SIGS Publications

Web and Internet

comp.lang.smalltalk – newsgroup

Software

Appendix 1 - Check Boxes, Radio Buttons, Input Fields, and their applications

Overview

Check boxes are GUI widgets used to turn an option on or off. Radio buttons have a similar purpose but they are used to select one of many options. If the number of options is large, variable, or unknown, single or multiple selection lists are preferable to buttons and boxes.

Check boxes are used independently of one another. Each check box has its own *Aspect* variable which holds a Boolean value in a *ValueHolder* and this value determines whether the box is on or off. Radio boxes, on the other hand, are used as groups and the whole group shares a single *Aspect* variable holding a *Symbol* in a *ValueHolder*. Each radio button has its own special symbol and when the button is selected, this symbol is stored in the group's *Aspect* variable.

Input fields behave like one-line text editors except that the text in the field can be accepted not only by the *accept* command but also by pressing the <Enter> key. The accepted value is stored in the *Aspect* variable.

A.1.1 Check boxes and radio buttons - an introduction

Built-in VisualWorks widgets include four kinds of buttons - action buttons, radio buttons, check boxes, and menu buttons. Action buttons are covered in Chapter 6 and we will now introduce check boxes and radio buttons (Figure A.1.1).

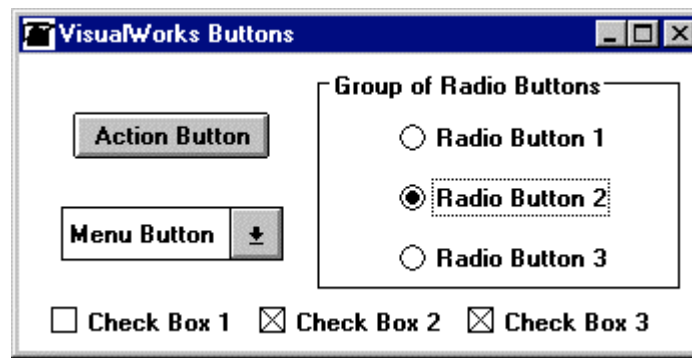


Figure A.1.1. Four built-in VisualWorks buttons - action button, check box, menu button, radio buttons.

Before we start dealing with the technical aspects of the new buttons, we will first comment on their role in user interfaces. This is an important aspect of interface design because all widgets have their established uses and uncommon use may confuse the user. The conventional uses of buttons are as follows:

- Action buttons are used to send messages, radio buttons and check boxes are used to select options.
- Radio buttons are used in groups and their control is designed so that exactly one radio button is on at any time. In other words, radio buttons are used to make 1-of-n related but mutually exclusive choices. Radio buttons are suitable when the number of options is relatively small; when the number of choices gets large or when it is calculated at run time it is usually preferable to use scrollable single-selection lists. The purpose of radio buttons is similar to that of menus which have the advantage that they require less window space. On the other hand, radio buttons show all available choices at all times whereas menus display choices only when they are activated.
- Check boxes are used to make individual independent choices. When several check boxes are available, any number of them may be selected at the same time. In other words, check boxes are used

to make m-of-n choices. Check boxes are suitable when the number of choices is relatively small. For a large number of choices, it is usually better to use scrollable multiple-selection lists.

Although almost all applications use buttons in the way described above, these conventions are not cast in stone or enforced by software and may be violated if there is a good reason.

Main lessons learned:

- GUI widgets are used according to established conventions. Ignoring these conventions may make the user interface difficult to use.
- Built-in VisualWorks buttons include action buttons, radio buttons, and check boxes.
- Action buttons are used to send messages.
- Radio buttons are used in groups to allow selection of one of several choices. When the number of choices gets large or when it is not fixed, use list widgets or menus instead.
- Check boxes are used in groups to make any number of simultaneous independent selections. When the number of choices gets large or when it is not fixed, use multiple-selection lists or menus.

A.1.2 Check boxes

The aspect of a check box is a ValueHolder containing a Boolean object. Its value is true when the box is on, and false when the box is off. The default initial value is false (the default setting of the box is off) but this can be changed programmatically or with the Properties Tool. The value holder is the widget's model, and the widget is the value holder's dependent. The relationship between the model and the check box is bi-directional: Clicking the box changes the value of the model, changing the value of the model changes the visual form of the check box.

The name of the value holder's instance variable (and the name of the message that accesses it) is the *Aspect* of the check box and it must be specified in the Properties Tool (Figure A.1.2). The label and several additional properties are optional.

The definition of both the *Aspect* instance variable and the *Aspect* accessing method can be created automatically with the *Define* command and no further modifications of the *Aspect* method are required. As a point of style, we like to derive the name of the *Aspect* from the label, as in our illustration. This makes it easier to remember which *Aspect* method belongs to which check box.

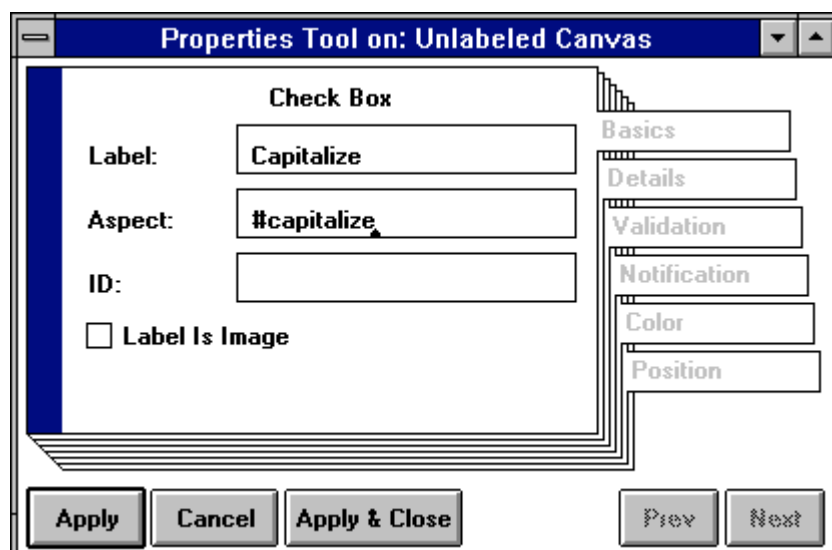


Figure A.1.2. The properties of a check box. If the label is specified as an image, the contents of the Label field must be the name of the message that returns the image that will be displayed next to the box.

Example: Math table with check boxes

Problem: Design and implement an application with the user interface in Figure A.1.3. Clicking *Display* prints a table of values of mathematical functions selected by the check boxes; values of arguments range from 1 to 20 in steps of 1. Any number of functions may be selected simultaneously but when no check box is on, clicking *Display* does not have any effect. (This is an appropriate use of check boxes because the application allows any number of selections.)

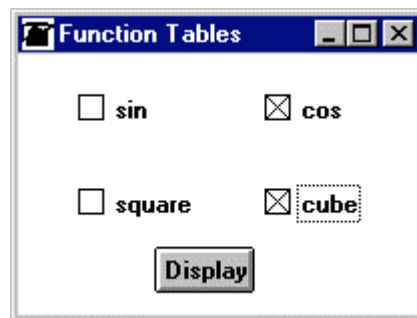


Figure A.1.3. Example: Math table application.

Solution: This simple application can be implemented as a single application model class and the only instance variables needed are those required by the check boxes. The work is done by the method activated by the *Display* button, and the corresponding method checks the current values of the value holders of all check boxes and produces the appropriate output to the Transcript.

Implementation: After painting the user interface and installing it on an application model, we defined the aspect methods and created the stubs using *Define*. To avoid confusion, we used the same names for button *Aspects* and for check box labels. The *Define* command created *Aspect* methods such as

```
square
    ^square isNil    ifTrue: [square := false asValue]
                    ifFalse: [square]
```

performing lazy initialization with *false* as the default value. If we wanted a check box to be initially on, we would have to initialize its *Aspect* variable in the initialization method to be a value holder on *true*, or specify that the box should be initially on in the Properties Tool.

The only remaining definition is that of method *display* sent by the action button. This method checks the values of the value holders of the check boxes and prints a table of the selected function values in the Transcript. Note that we use message value to extract the Boolean value from the *Aspect* variables:

```
display
"Check which boxes are on, calculate corresponding values, and display them in Transcript. Don't print anything if no box is selected."
(sin value or: [cos value or: [square value or: [cube value]]])
    ifFalse: [^self].
"For all values in the range, calculate selected functions and print in Transcript."
Transcript clear.
sin value ifTrue: [Transcript show: 'sin'; tab].
cos value ifTrue: [Transcript show: 'cos'; tab].
cube value ifTrue: [Transcript show: 'cube'; tab].
square value ifTrue: [Transcript show: 'square'; cr].
1 to: 20 do: [:argument | "Examine all check boxes to determine which functions to print."
    Transcript show: argument printString; tab.
    sin value ifTrue: [Transcript show: argument sin printString; tab].
    cos value ifTrue: [Transcript show: argument cos printString; tab].
```

```
cube value ifTrue: [Transcript show: (argument squared * argument) printString; tab].  
square value ifTrue: [Transcript show: argument squared printString; cr]]
```

The application works except for a small problem that occurs when you click *Display* and the *square* button is not selected. We leave it to you to identify and correct this minor problem.

Main lessons learned:

- The *Aspect* of a check box is the name of an instance variable holding a Boolean ValueHolder representing the state of the box. It is also the name of the accessing method of this variable.
- The *Aspect* method and variable of a check box can be automatically defined with *Define* and the method does not require any modification if the box is initially off. Use the Properties Tool to turn the box initially on.

Exercises

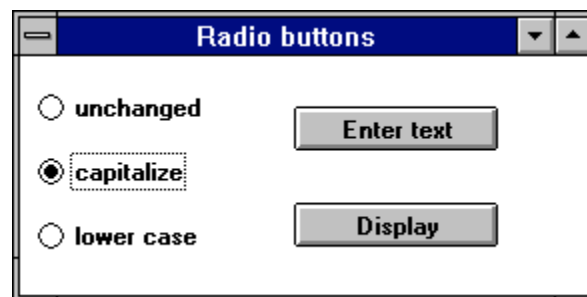
1. Implement the example from this section.
2. Our definition of display tests all check boxes on each pass and this is quite inefficient. Can you find a better solution?

A.1.3 Radio buttons

Radio buttons are always used in groups and exactly one radio button is always selected¹. Because of this, all buttons in the group share one *Aspect* variable and its value identifies the button that is currently selected. This is done by assigning one *Selection* value to each button via the Property tool. *Selection* acts like a symbolic tag of the button and the value in the *Aspect* variable identifies the radio button that is currently on. Each radio button thus requires an *Aspect* variable and its accessing method (shared by all buttons in a group), and a *Selection* value (unique for each check box in a group). If the window contains several groups of radio buttons, the scope of *Selection* values is restricted to each group. As a consequence, the same *Selection* value may be used by different radio buttons if they are not in the same group (don't share the same *Aspect* variable).

Example: Controlling the case of a string

Problem: Design an application with the user interface in Figure A.1.4. Clicking *Enter text* opens a dialog asking the user to enter a string. Clicking *Display* prints the string in the Transcript using the style selected by the setting of the radio buttons as follows: When the *unchanged* radio button is on, the text is displayed unchanged, when the *capitalize* button is selected the text is displayed capitalized, when the *lower case* button is selected the text is displayed in lower case. (This is an appropriate use of radio buttons because exactly one selection must be made at any time.)



¹ If you don't initialize the *Aspect* value holder, the window will open with no radio button selected. This should normally be changed because it is inconsistent with the normal behavior of radio buttons.

Figure A.1.4. Example: Radio buttons used to select one of three display styles.

Solution: The problem can again be implemented using just an application model class. We need one instance variable to hold the text entered by the user and one instance variable for the *Aspect* of the group; both the *Aspect* variable and its accessing method can be automatically created by *Define*.

We selected the name *#case* for the *Aspect* of the three radio buttons because its value determines how the text will be displayed. For the *Select* parameter of individual buttons, we chose *#unchanged*, *#capitalize*, and *#lowerCase* - names derived from button labels.

The accessing method automatically created by *Define* turns all radio buttons initially off. To turn the *capitalize* button on, we can initialize *style* to *#capitalize* as follows:

```
initialize  
    case := #capitalize asValue
```

Alternatively, if we defined a *case* accessing method by the *Definer*, it has an accessor with lazy initialization and we can then initialize the value with

```
initialize  
    self case value: #capitalize
```

We leave the method for the *Enter text* button to you as an exercise. It should open a dialog, prompt the user for a string, and save its value in an instance variable of the application model; we will call this variable *text*. The only remaining method is *display*. This method checks which radio button is on and displays the text in the Transcript using the selected style:

```
display  
"Display text using selected style but leave initial text unchanged."  
| newText |  
"Calculate text using selected style."  
newText := case == #unchanged  
    ifTrue: [text]  
    ifFalse: [case == #capitalize  
        ifTrue: [text asUppercase]  
        ifFalse: [text asLowercase]].  
  
"Display in Transcript."  
Transcript cr; show: newText
```

This seems perfectly reasonable - but it does not work! The text is always displayed in lower case, no matter which style we choose. Obviously, something must be wrong with the value of variable *newText*. And indeed - we made the typical mistake with value holders: The value of *case* is a *ValueHolder* - not a *Symbol* - and the value is 'inside' it. Expression

```
case == #unchanged
```

checks whether the variable holds a *Symbol*, which it does not. All checks thus fail and the last alternative of the nested messages is always executed, changing the text to lower case. For proper operation, we must check what is the *value* held in the *case ValueHolder*, not what is the value of *case* itself. The following version is correct:

```
display
| newText |
newText := case value == #unchanged
    ifTrue: [text]
    ifFalse: [case value == #capitalize
        ifTrue: [text asUppercase]
        ifFalse: [text asLowercase]].

Transcript cr; show: newText
```

Main lessons learned:

- Radio buttons are used in groups to select exactly one of several options.
- For each radio button, specify at least *Selection*, and *Aspect*.
- The *Selection* value of a button is a Symbol assigned to the *Aspect* when the button is clicked. Each button in a group must have its own unique *Selection* value.
- The *Aspect* of a radio button is the name of an instance variable holding a ValueHolder on a Symbol; it is also the name of the accessing method of this variable.
- The value of the *Aspect* variable is the value of the *Selection* property of the currently selected button.
- The *Aspect* name must be identical for all radio buttons in a group.
- The *Selection* value of the radio button which should be on when the window first opens must be assigned as the value of a ValueHolder to the *Aspect* variable of the group in the initialize method.

Exercises

1. Implement the example from this section.

A.1.4 Input fields

An input field is essentially a one-line text editor complete with an <operate> menu. When set to read-only operation it can function as a program-controlled label.

Each input field has an *Aspect* variable containing a ValueHolder with a string. When the user enters text into the field and hits <Enter> or selects *accept* from the <operate> menu, the field stores its contents in its *Aspect* variable. Before the user accepts the contents of the text field in this way, the text displayed in the field and the text stored in its *Aspect* variable may be different and the text field widget itself thus has another instance variable which holds the displayed text (Figure A.1.5). In this respect, the input field is again similar to the text editor.

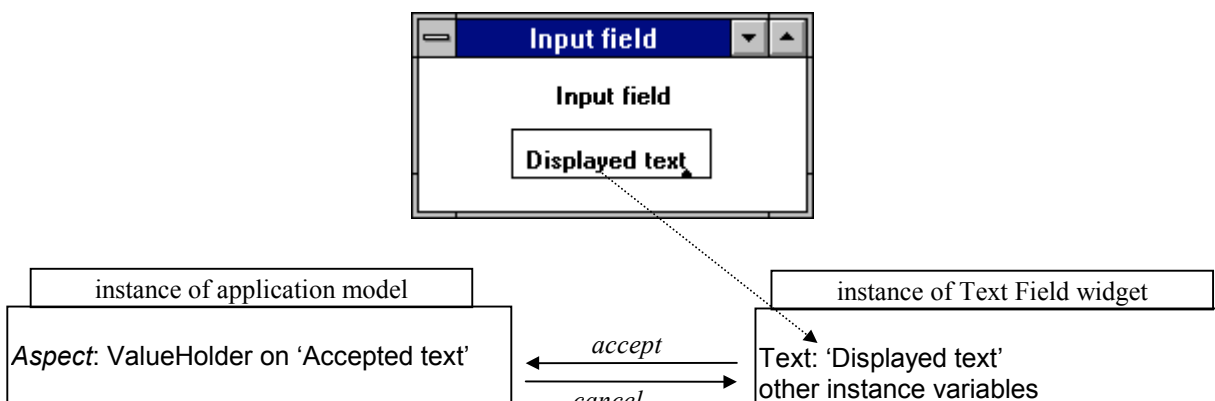


Figure A.1.5. The displayed and the accepted text may be different and are held in two distinct variables.

Example 1: Read-write input field connected to a read-only field

Problem: Create a window with two input fields as in Figure A.1.6. The top field is for input and the bottom field is for output. When the user enters text into the top field and *accepts* it, the text is displayed in the bottom field. The bottom field is a read only field (and the name *input field* is thus a misnomer).

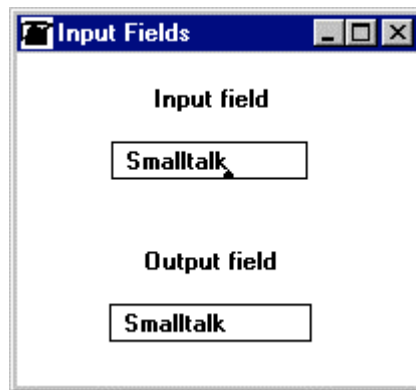


Figure A.1.6. Example: Accepted changes in the top field are displayed in the read-only field below.

Solution: Paint the widgets and specify their *Aspects* (we chose *inputField* for the top field and *outputField* for the bottom field), select *Read Only* as the special property of the bottom field. Install the canvas and use the *Define* command to create the *Aspect* variables and methods of both fields. Both methods perform lazy evaluation, assigning

String new asValue

to *Aspect* variables when first accessed. You can now open the application and check that you can type into the top field but not into the bottom field. Of course, typing into the top field does not have any effect on the bottom field because we have not defined any mechanism for linking the two fields together.

The implementation of the desired behavior will be as follows: When the user *accepts* new text in the input field and thus changes the value of variable *inputField*, the change of *inputField* will send message *newInput* to *outputField*; this message will send the value: message to *outputField* and change its value. This, in turn, will change the text displayed in the output field because the output field widget is a dependent of the *outputField* variable. In plain English, when the user tells *inputField* 'change your value', *inputField* will tell *outputField* 'change your value'; since the output field widget is a dependent of *outputField*, this will say 'redisplay yourself with the new text' to the *outputField* widget (Figure A.1.7).

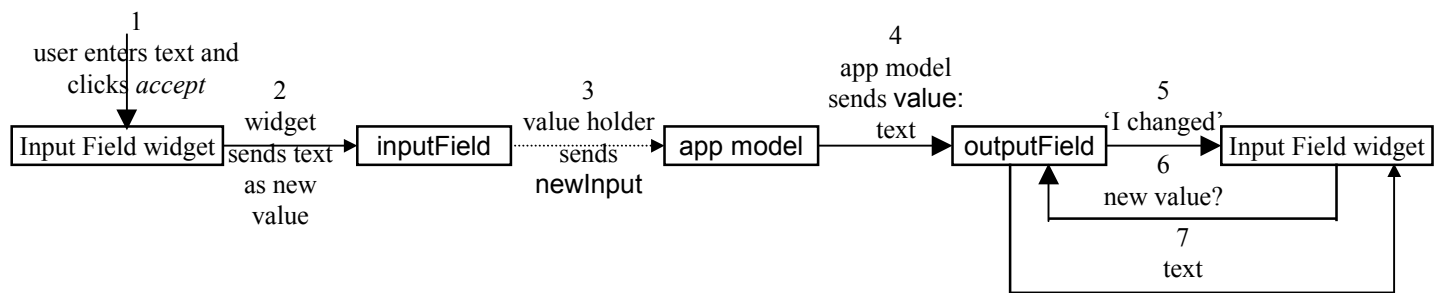


Figure A.1.7. Desired link between the two text fields. Full lines represent built-in behavior, interrupted line is behavior that must be defined.

To set this mechanism up, we must

1. inform the *inputField* value holder that when its value changes, it must send message *newInput* to the application model.

2. write method `newInput`.

To implement Item 1, we must send `onChangeSend:to:` to the model of the widget in the `initialize` method²:

initialize

```
inputField onChangeSend: #newInput to: self
```

Unfortunately, this does not work and we get the exception in Figure A.1.8 even before the window opens.

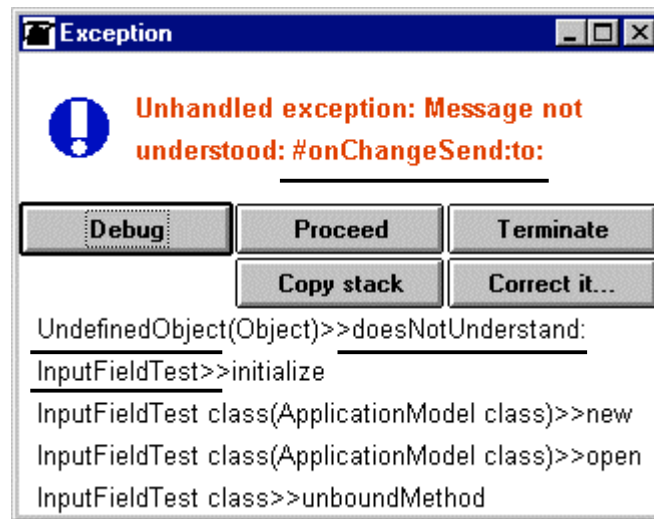


Figure A.1.8. Error notifier obtained when trying to open the example application.

We don't even need to open the debugger to find the reason for this problem - the cause is clear from the notifier which says that "while executing method `initialize`, our application model sends message `onChangeSend:to:` to an `UndefinedObject`". Why `UndefinedObject`? In the definition of `initialize`, we are sending `onChangeSend:to:` to `inputField` but `inputField` is still `nil`. To make `inputField` a `ValueHolder` on the string in the input field widget, we must first assign a `ValueHolder` object to `inputField`:

initialize

```
inputField := " asValue. "This value will be displayed when the window opens."  
inputField onChangeSend: #newInput to: self
```

With this modification, the window opens properly and whenever the user *accepts* new text in the input field, `inputField` sends `newInput` to `self` - the application model.

We have now established the 'link' and to complete the program, we must write the `newInput` method. When this method is invoked it should simply get the new text (the value of `inputField`) and assign it to the value of `outputField`. The definition is as follows:

newInput

```
outputField value: inputField value
```

The example is now fully operational.

The mechanism behind `onChangeSend:to:`

² Alternatively, we could specify `#newInput` as the *Action on changed* of the widget in the Notification property in UI Painter as explained in Appendix 8.

We have seen that the result of an expression such as

`aValueHolder onChangeSend: aMessage to: aReceiver`

is that whenever `aValueHolder` gets the `value:` message (for example by a mechanism built into the input widget), it sends `aMessage` to `aReceiver`. To see how this works, we examine the definition of `onChangeSend:to:` in class `ValueModel` and find

onChangeSend: aSymbol to: anObject

"Arrange to receive a message with aSymbol when the value aspect changes on anObject."

```
self    expressInterestIn: #value
      for: anObject
      sendBack: aSymbol
```

where the definition of `expressInterestIn:for:sendBack:` in class `Object` is

expressInterestIn: anAspect for: anObject sendBack: aSelector

"Arrange to receive a message with aSelector when anAspect changes at anObject"

```
| dt |
dt := DependencyTransformer new.
dt    setReceiver: anObject
      aspect: anAspect
      selector: aSelector.
self addDependent: dt
```

This shows that

`aValueHolder onChangeSend: aMessage to: aReceiver`

creates a `DependencyTransformer` and adds it to the dependents of `aValueHolder`. As a consequence, when `aValueHolder` gets `value:`, it tells this new dependent about it.

What is a `DependencyTransformer` and what does it do when it is notified by its model `ValueHolder`? When you examine `DependencyTransformer`, you will find that it keeps the selector of a message (`aSelector` in the definition above) and a pointer to another object (`anObject`). When it receives message `update:` with argument `anAspect`, it sends message `aSelector` to `anObject`. In summary, the `onChangeSend:to:` sets up a new dependent of the value holder, and this new dependent fires its assigned message to its assigned receiver whenever asked to update itself (Figure A.1.9). Essentially, a `DependencyTransformer`'s is a translator that transforms the `update:` message into another message.

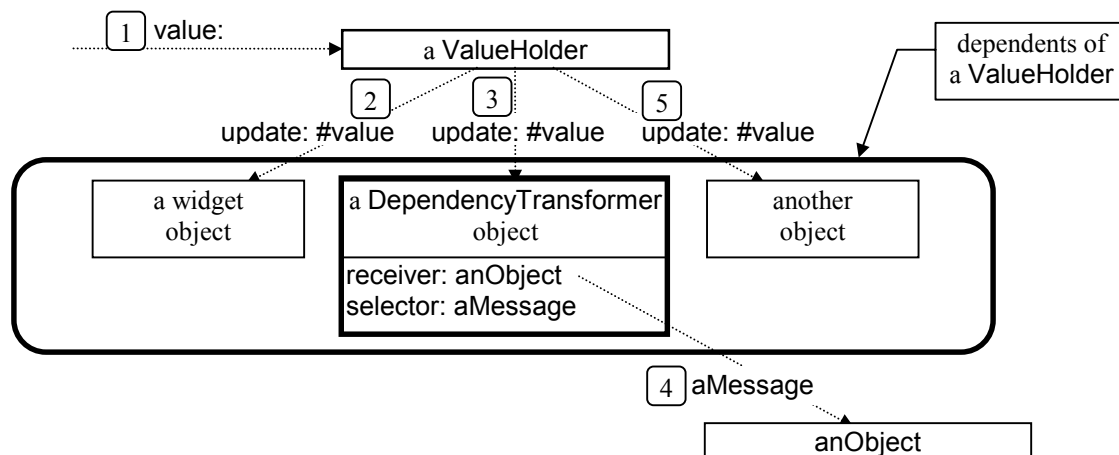


Figure A.1.90. Events triggered by sending `value:` to a `ValueHolder` with a dependent `DependencyTransformer`.

Main lessons learned:

- An input field is a one-line text editor.
- An input field has only one essential property - its *Aspect*. The *Aspect* is the name of an instance variable holding a *ValueHolder* on the accepted text, and the name of its accessing method.
- To display text when the window opens, assign a *ValueHolder* with this text to the *Aspect* variable in the *initialize* method.
- To store the text in an input field to the *Aspect* variable, the user must press <Enter> or execute *accept* from the field's <operate> menu.
- Since the accepted value and the displayed value may be different, the displayed value is held in an instance variable of the input field; the accepted value is held in the *Aspect* variable in the application model.
- An input field may be defined as a read-only widget. This means that the user cannot type into it and that the field is insensitive to mouse button clicks, making it impossible to give it a menu.
- To change the text displayed in an input field programmatically, send *value:* to its *Aspect* variable.
- To force a variable holding a *ValueHolder* to send a message whenever its value changes, send it the *onChangeSend: aMessage to: anObject* message. *anObject* is normally *self* because *aMessage* is usually defined in the application model.
- The *onChangeSend: aMessage to: aReceiver* message is the basis for setting up links between widgets.
- The object that transforms the change of a value holder into a message to an object is an instance of *DependencyTransformer*.

Exercises

1. Modify our example 1 to display 'Input text' in the input field and 'Output text' in the output field when the window first opens. Note that once you do this, you don't need lazy evaluation for this variable any more.
2. Modify our example as follows: Change the type of input/output to numeric, center the text in the field, change the color of the input field text to blue and that of the output field to red.
3. Write a short description of *DependencyTransformer*.
4. Trace the behavior of the *DependencyTransformer* by including a *self halt* into the *newInput* method. Write a description of your findings.
5. Enact a typical scenario for the input field - output field example.

A.1.5 A computerized restaurant menu

In this section, we will design and implement a simple application using check boxes and an input field - a computerized restaurant menu that allows the user to choose meals by clicking buttons (Figure A.1.10). When the user selects or deselects an item, the menu immediately updates the current total at the bottom of the window. When the user presses the OK button, a window opens saying that the order is being processed. Check boxes for selection of pizza toppings are enabled only when the *Basic* pizza check box is on.

Category	Item	Price
Soup	<input checked="" type="checkbox"/> Gazpacho	3.50
	<input type="checkbox"/> Lentil soup	2.50
Pizza	<input checked="" type="checkbox"/> Basic	3.50
	Toppings <input type="checkbox"/> Mozzarella	1.15
	<input checked="" type="checkbox"/> Figs	1.25
	<input type="checkbox"/> Pineapple	1.00
Entrees	<input type="checkbox"/> Halibut on Almonds	6.95
	<input checked="" type="checkbox"/> Chicken Ivanek	7.25
Deserts	<input type="checkbox"/> Jana's apple pie	3.70
	<input checked="" type="checkbox"/> Ondrej's crepes	4.10
	<input type="checkbox"/> Dominika's brownies	2.50
Beverages	<input type="checkbox"/> Coke	0.80
	<input checked="" type="checkbox"/> Apple juice	0.70
Total		20.30

OK

Figure A.1.10. Computerized restaurant menu.

Design: Although the problem could be a part of a large application that keeps track of bills, supplies, and other things, we will treat it as an exercise in UI design and implement it with a single application model class. *Aspect* methods of individual check boxes and the text field will be created automatically, the *Total* will be a read-only field with centered output, and the *OK* button will require a method that will open the confirmation window. An initialization method will set up the dependencies.

Implementation: Painting and installing the interface and defining *Aspects* is, in general, quite routine except that the value in the input field is a number and should be displayed with two decimal digits (as dollars and cents). Although input fields display strings, other kinds of objects such as numbers and dates are also frequently required and the text field widget can convert them to and from strings automatically. In our case, we select the *fixed point* format in the Properties Tool window. We also select *centered* format.

The next step is an initialization method defining mainly how the application model responds to state changes of individual check boxes. As we know, registering interest in changes is achieved by the *onChangeSend:to:* message and this requires that we make the value of each *Aspect* variable a *ValueHolder*. Since we have 13 check boxes, this will require 13 *false asValue* messages followed by 13 *onChangeSend:to:* messages. This would make the definition much longer than the recommended method size (as a rule of thumb, method definitions should not be longer than the browser's text field) and we will thus divide it into the variable initialization part and the change specification part. Each part will be implemented by a separate method:

initialize

```
self initializeVariables.  
self initializeChanges
```

The definition of initializeVariables is

initializeVariables

"Set check box Aspect variables as value holders on false to be able to send them onChangeSend:to: during the next step of initialization, and to set all check boxes off."

```
appleJuice := false asValue.  
applePie := false asValue.  
basic := false asValue.  
brownies := false asValue.  
chicken := false asValue.  
coke := false asValue.  
crepes := false asValue.  
figs := false asValue.  
gazpacho := false asValue.  
halibut := false asValue.  
lentil := false asValue.  
mozzarella := false asValue.  
pineapple := false asValue.  
total := 0.0 asValue
```

"To display 0.0 for total when the window opens"

and method initializeChanges is

initializeChanges

"Specify change messages to be sent on changes of value holder variables"

"Each change message will check the state of the value holder and update the total."

```
appleJuice onChangeSend: #checkAppleJuice to: self.  
applePie onChangeSend: #checkApplePie to: self.  
basic onChangeSend: #checkBasic to: self.  
brownies onChangeSend: #checkBrownies to: self.  
chicken onChangeSend: #checkChicken to: self.  
coke onChangeSend: #checkCoke to: self.  
crepes onChangeSend: #checkCrepes to: self.  
figs onChangeSend: #checkFigs to: self.  
gazpacho onChangeSend: #checkGazpacho to: self.  
halibut onChangeSend: #checkHalibut to: self.  
lentil onChangeSend: #checkLentil to: self.  
mozzarella onChangeSend: #checkMozarella to: self.  
pineapple onChangeSend: #checkPineapple to: self
```

This does not appear too intelligent because both methods contain a long sequence of essentially identical messages. We will examine whether there are better solutions in the next section.

The *Action* method of the OK button is trivial - it simply opens a Dialog box - and we leave it to you as an exercise. The *Aspect* methods of the check boxes created by *Define* should be created with *initialization* off (we already initialized all *Aspect* variables) and the only methods that require attention are the check messages in initializeChanges. Most of them will be defined along the following pattern:

checkAppleJuice

"The apple juice box has just changed its state. If it has been clicked *on*, add the price of the item to total, if it has been clicked *off*, subtract the price from the total."

```
appleJuice value  
  ifTrue: [total value: (total value + 0.70) ]  
  ifFalse: [total value: (total value - 0.70) ]
```


The logic of the definition captures the fact that the method is executed for *every* change of the state of the check box - when the box is clicked on as well as when it is clicked off. Note that we use `value:` to set the new total in order to propagate the change to the input field widget.

The only change method that is different is the method for the Basic Pizza check box. This method must perform the following tasks:

- If the box is clicked on, it must change the value of the total and enable the toppings buttons.
- If the box is clicked off, it must disable the toppings buttons and subtract the price of the pizza *and* the price of any toppings that have been on from the total.

To implement this behavior, we must know how to enable and disable a widget. The principle is that if you give a widget an ID, you can access it at run time via the UI builder, for example to enable or disable it (messages `enable` and `disable`) or to hide or show it (`beInvisible` and `beVisible`). An ID is assigned via the Properties tool as in Figure A.1.11.

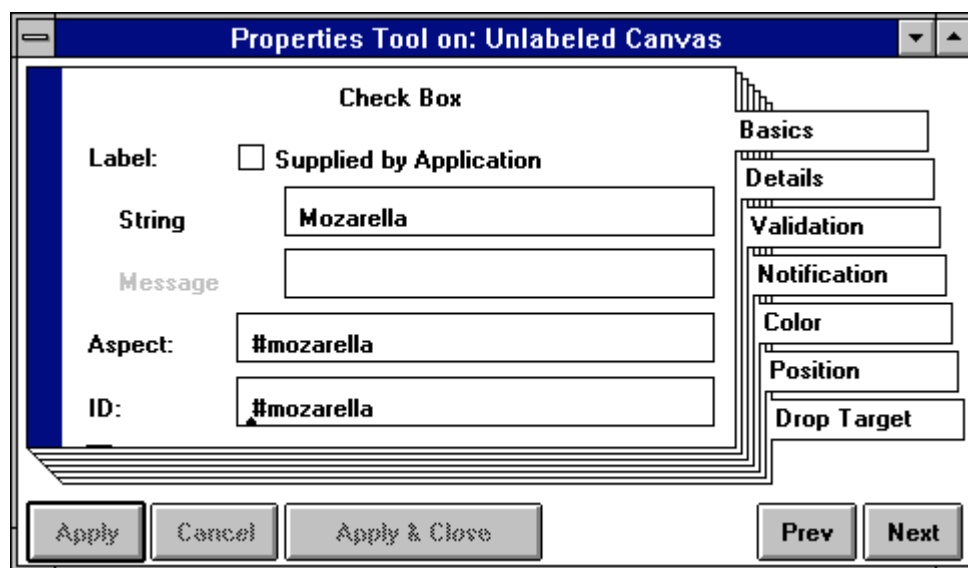


Figure A.1.11. Toppings boxes must have IDs so that the program can enable and disable them.

We have just established the principle of the logic of method `checkBasic`. The details are as follows:

```
If Basic box is on then
    add basic price to total
    enable toppings boxes
If Basic box is off then
    disable toppings boxes
    subtract basic price from total
    change the value of those toppings boxes that are on to off using the value: message
    (The corresponding value holders will send their change messages which will
     turn toppings boxes off and subtract their price from total)
```

From this specification, we can now easily write the definition:

checkBasic

"The Basic box has just changed, recalculate the total and modify the display"

basic value

```
ifTrue: "Box was just clicked on - enable toppings boxes, add price of basic pizza to total."
[total value: total value + 3.50.
```

```
(builder componentAt: #mozzarella) enable.  
(componentAt: #figs) enable.  
(builder componentAt: #pineapple) enable.  
(builder componentAt: #mozzarellaPrice) enable.  
(builder componentAt: #figsPrice) enable.  
(builder componentAt: #pineapplePrice) enable]  
ifFalse: "Box was just clicked of - disable toppings boxes, subtract price of basic pizza from total."  
[(builder componentAt: #mozzarella) disable.  
(builder componentAt: #figs) disable.  
(builder componentAt: #pineapple) disable.  
(builder componentAt: #figsPrice) disable.  
(builder componentAt: #pineapplePrice) disable.  
total value: total value - 3.50.  
"Subtract price of selected toppings from total."  
mozzarella value ifTrue: [mozzarella value: false].  
figs value ifTrue: [figs value: false].  
pineapple value ifTrue: [pineapple value: false]]
```

This definition shows how to communicate with widgets at run time: To obtain a widget³, ask the builder using componentAt: ID; then send the appropriate message.

How widget IDs work

The basis of the operation of IDs is method componentAt which is defined in class UIBuilder as follows:

componentAt: aKey

"Retrieve the SpecWrapper of the indicated component."
^namedComponents at: aKey ifAbsent: [nil]

This definition shows that to the builder gets a widget's wrapper by asking its instance variable namedComponents for the component at the given ID. namedComponents holds a registry of ID -> widget wrapper pairs and we encourage you to examine it with an inspector.

Main lessons learned:

- Any GUI widget may have an ID, an instance of Symbol. Widget IDs must be unique within an application model or more accurately within the scope of a UI builder.
- An ID is required only for direct communication between the application and the widget, for example to enable, disable, show, or hide it.
- The builder holds a registry of widget IDs. To obtain a widget (or rather its wrapper), request it via its ID from the builder using the componentAt: id message. The builder itself is an instance variable of the application model.
- A method definition should not exceed the size of the browser's code view by much. If it does, try to find a more compact solution or split the method into several shorter methods. There are, of course, justifiable exceptions to this rule.

Exercises

1. Complete the Restaurant Menu application and test it. You will have to learn how to use the region widget to draw the rectangles around the menu. Make the rectangles blue with a red outline, display the names of the courses such as *Soup* and *Pizza* in red, and the rest in black.
2. Equip the window with a vertical scroll bar to provide more space for courses.

³ In reality, the builder does not return the widget itself but the 'wrapper' around it but this distinction is unimportant at this point.

3. Implement meal names with read-only input fields with no borders instead of labels. What are the relative advantages of the two implementations?
4. Browse class `UIBuilder`, read its comment, and examine its `examples` class protocol. These example methods show how to create a builder and a user interface programmatically instead of using `UI painter`.
5. Insert a breakpoint into `checkBasic` to inspect the components of the builder at run time.

A.1.6 Other implementations of Restaurant Menu

When we introduced the `initialize` method in the previous section, we noted that both of its constituent messages are awkward because they contain too much repetitive code. As an example,

`initializeVariables`

```
appleJuice := false asValue.  
applePie := false asValue.  
basic := false asValue.  
brownies := false asValue.  
chicken := false asValue.  
coke := false asValue.  
crepes := false asValue.  
figs := false asValue.  
gazpacho := false asValue.  
halibut := false asValue.  
lentil := false asValue.  
mozzarella := false asValue.  
pineapple := false asValue.  
total := 0.0 asValue
```

consists of a sequence of identical assignments to different variables. One would think that it should be possible to rewrite it more compactly, for example as

`initializeVariables`

"Initialize all *Aspect* variables as value holders on false to set all check boxes initially off."

```
appleJuice applePie := basic := brownies := chicken := coke := crepes := figs := gazpacho :=  
    halibut := lentil := mozzarella := false asValue.  
total := 0.0 asValue
```

"To display 0.0 for total when the window opens"

Although this solution is legal, it is incorrect because it assigns the same value holder to all *Aspect* variables. As it is, changing the state of any check box will thus change all others (Figure A.1.12).

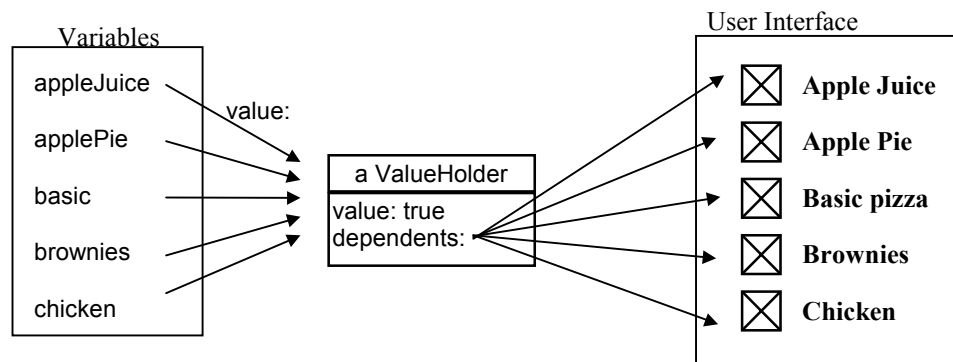


Figure A.1.12. Assigning the same `ValueHolder` to all *Aspect* variables locks all check boxes together.

We must assign a different value holder to each variable (Figure A.1.13) along the following lines:

`initializeVariables`

“Initialize all Aspect variables as value holders on false to turn all check boxes off.”
 #(appleJuice applePie basic brownies chicken coke crepes etc.) do:
 [:variable | variable := false asValue].
 total := 0.0 asValue “To display 0.0 for total when the window opens”

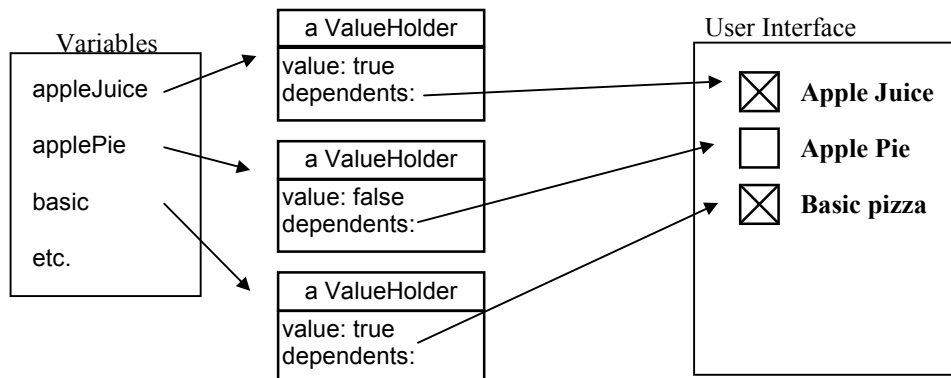


Figure A.1.13. Each check box must have its own ValueHolder.

This formulation will not work either because the literal array is an array of Symbols and consequently the value of the block variable *variable* is a Symbol. This is unacceptable because the left hand side of an assignment must be a variable name. Since aspect variables are necessary and assignment to a variable cannot be avoided, it seems that we cannot simplify the *initializeVariables* method. But what about the *initializeChanges* method?

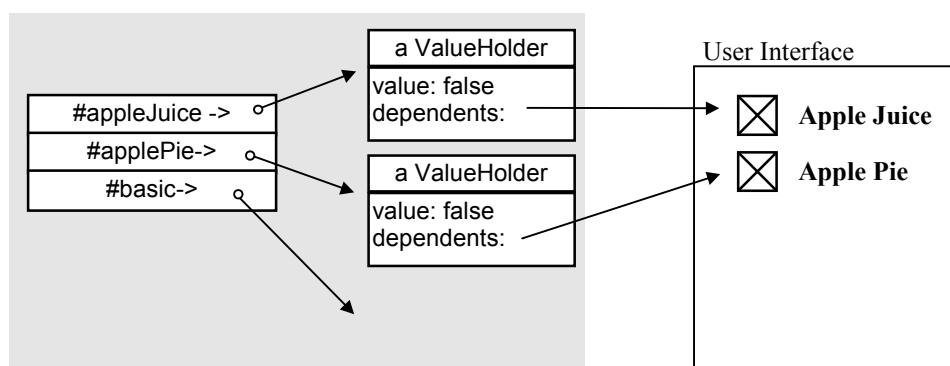
To find a working solution, we must return to the origin of our problem and consider what we are trying to do in *initializeChanges*. Our ultimate goal is to send *onChangeSend:to:* to the value holders associated with the individual check boxes. In other words, the point is not to communicate with the *variables* but to use them to access the *value holders*. If there is some other way to access the value holders directly, we don't need to bother with the variables. And if we don't have to access the variables, then there may be an easier way.

It turns out that in our case there are two ways to access the value holders directly - one through their accessing methods and one through the builder. We will now formulate a new solution using both approaches.

Solution 1: Using the builder to access a widget's value holder

To access the value holder of a widget through the UI builder requires an understanding of the application opening process. From the perspective of our current interest, this process works as follows:

1. When the application model class gets the open message or its equivalent, it creates an instance of a *UIBuilder* with, among other things, variable bindings containing an *IdentityDictionary*.
2. The builder reads the specification of the interface (in our case stored in *windowSpec*), extracts the description of each widget, and executes the following steps if the widget has an *Aspect* method:
 - i. It sends the component's *Aspect* message to the application model, expecting to get the value of the widget's *Aspect* variable, a *ValueHolder*.
 - ii. It adds the pair
 (*Aspect* name as Symbol) -> *ValueHolder*
 to the bindings dictionary.
 - iii. It makes the corresponding widget a dependent of the *ValueHolder* (Figure A.1.14).



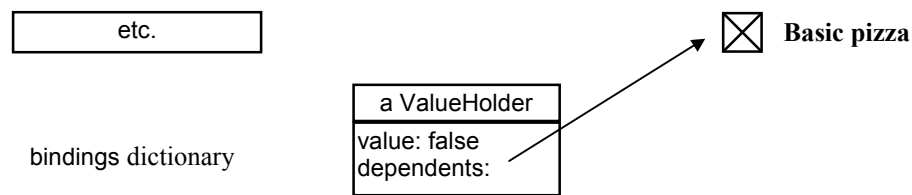


Figure A.1.14. Builder's bindings dictionary.

Once the bindings dictionary exists, we can access its values (the value holders) by sending message `aspectAt: aSymbol` to the builder. As an example, `aspectAt: #basic` returns the value holder of the *Basic* check box.

We conclude that if we leave the `onChangeSend:to:` message until *after* the builder is finished building the interface, we can remove the `initializeChanges` method and access the value holders via the builder instead of going through the variables as in

postBuildWith: aBuilder

"Send `onChangeSend:to:` to each aspect value holder."

`(aBuilder aspectAt: #appleJuice) onChangeSend: #checkAppleJuice to: self.`

`(aBuilder aspectAt: #applePie) onChangeSend: #checkApplePie to: self.`

`etc.`

This definition does not look any simpler than `initializeChanges` but it can be simplified: We can now put all *Aspect* Symbols into an array, put the corresponding change message symbols in another array so that names of *Aspect* symbols parallel symbols of their corresponding change messages (Figure A.1.15), and all messages in a single enumeration statement.

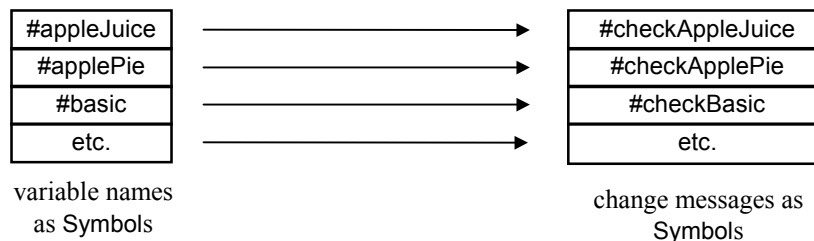


Figure A.1.15. Parallel arrays of variables as symbols and their corresponding change messages.

The definition can now be written as follows:

postBuildWith: aBuilder

"Send `onChangeSend:to:` to each aspect value holder."

`aspectSymbols := #(#appleJuice #applePie #basic #brownies etc.).`

`changeSymbols := #(#checkAppleJuice #checkApplePie #checkBasic #checkBrownies etc.).`

`aspectSymbols with: changeSymbols do[:aspectSymbol :changeSymbol]`
`(builder at: aspectSymbol) onChangeSend: changeSymbol to: self]`

Solution 2: Accessing value holders via their accessing methods

Another way to register change messages is as follows: Since we have *accessing methods* for the value holders, we can put their names in an array as Symbols, and execute them using enumeration and `perform:`, obtaining the value holders. The principle is that an expression such as

`self perform: #appleJuice`

executes

self appleJuice

and returns the value holder for the *Apple Juice* check box. We can then send `onChangeSend:to:` to this value holder as in

```
(self perform: #appleJuice) onChangeSend: #checkAppleJuice to: self
```

The corresponding definition is

postBuildWith: aBuilder

“Send `onChangeSend:to:` to each aspect value holder.”

```
aspectMethods := #(#appleJuice #applePie #basic #brownies etc.).
```

```
changeMethods := #(#checkAppleJuice #checkApplePie #checkBasic #checkBrownies etc.).
```

```
aspectMethods with: changeMethods do[ :aspectMethod :changeMethod|  
    (self perform: aspectMethod) onChangeSend: changeMethod to: self]
```

Several notes are in order:

- If we have object-accessing methods, we can use this approach in any situation where we need to access a collection of objects in a uniform way.
- With this approach, we don’t have to send `onChange` assignment in `postBuild:`; it will work equally well in `initialize` because of the operation of *Aspect* methods: When the builder builds the bindings dictionary, it sends the *Aspect* message and if the *Aspect* variables already contain value holders, the accessing message returns the value holder unchanged. The interface building process thus will not disrupt the bindings and the `onChange` message assignments built by our code.

The last note has a very important flip side: When using *Aspect* variables, one must be careful not to affect the bindings built by the builder. As an example, assume that we want to open our window with all the check boxes on. This requires that the values of the *Aspect* objects be value holders on `true` and this could be achieved for example by

postBuildWith: aBuilder

```
...  
    #(#appleJuice #applePie basic etc.) do:  
        [ :aspectSymbol | (builder at: aspectSymbol) value: true].  
...
```

It is tempting to write

postBuildWith: aBuilder

```
...  
    #(appleJuice applePie basic etc.) do:  
        [ :aspectSymbol | (builder at: aspectSymbol put: true asValue)].  
...
```

but this is wrong. The reason is that when the builder constructs the bindings, it makes each check box a dependent of a particular *ValueHolder*. Sending `value:` to this value holder changes its value but leaves the model→dependent binding between the value holder and the widget unchanged.

If we used `at:put:` instead of `value:`, the builder would replace the existent *ValueHolder* with a new *ValueHolder* and since a new value holder does not have any dependents, the dependency between the value holder and the widget would be broken. Future changes to the value holder and changes to the state of the widget would thus be independent of one another (Figure A.1.16). This mistake is quite common and leads to problems that are not easy to trace.

new
value holder

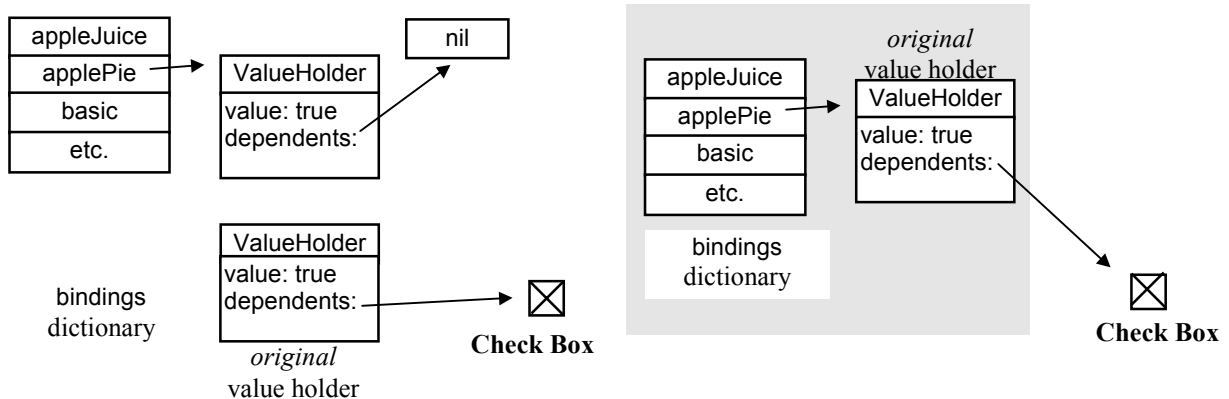


Figure A.1.16. Effect of builder *at: applePie put: true asValue* (left) and *(applePie at: symbol) value: true* (right). Since the original value holder on the left is eliminated from the bindings dictionary and garbage collected, unless other references to it exist.

Another improvement: Updating total by a single method

The use of one total-updating message for each check box as in

checkAppleJuice

```
appleJuice value
  ifTrue: [total value: (total value + 0.70) ]
  ifFalse: [total value: (total value - 0.70) ]
```

seems redundant because most of the check box total-updating messages are almost identical. Another way to solve the problem is to define a single method handling all updates (other than the pizza group which is different) and send it when any value holder changes. The method will go through all value holders, check their state, and recalculate the total as follows:

updateTotal

```
| newTotal prices values |
"Check all value holders and update the total."
values := #(appleJuice applePie etc.).
prices := #(0.70 3.70 etc.).
newTotal := 0.
values with: prices do: [ :id :price| (builder at: id) value "If the box is on, add item's price."
                                ifTrue: [ newTotal := newTotal + price]].
```

total value: newTotal

With this method, we can now change all check methods to the following format:

checkAppleJuice

self updateTotal

Considering that we must create the values and prices arrays every time when we execute updateTotal, it will be better to store values and prices in two new instance variables called values and prices and initialize them to the two arrays shown above once and for all. The complete solution is now as follows:

ApplicationModel subclass: #DriveIn
instanceVariableNames: 'prices values etc.'
etc.

initialize

values := #(appleJuice applePie etc.).
prices := #(0.70 3.70 etc.).
etc.

postBuildWith: aBuilder

values do: [:id | (builder at: id) onChange send: updateTotal to: self]

updateTotal

| newTotal prices values |
"Check all value holders and update the total."
newTotal := 0.
values with: prices do: [:id :price | (builder at: id) value "If the box is on, add item's price."
ifTrue: [newTotal := newTotal + price]].
total value: newTotal

and

checkAppleJuice

self updateTotal

and similarly for all other check methods.

In this solution, we are doing many more calculations but the solution is much simpler. Since the number of calculations is still very small and their nature trivial, this solution is better.

Main lessons learned:

- While building the user interface, **UIBuilder** constructs a dictionary of bindings containing *Aspect* Symbols as keys and their value holders as values. To obtain a value holder, it sends the *Aspect* message defined as the widget's Property to the application model. It then registers the widget's wrapper as the value holder's dependent.
- Accessing values of instance variables by enumeration can be simplified by defining an accessing method for each of them and enumerating over an array of Symbols corresponding to the selectors of these accessing messages.

Exercises

1. Complete the application.
2. Since all check methods do exactly the same thing if we used the updateTotal method, it does not seem that we need them after all. Modify the implementation accordingly.

A.1.7 Validation of user input

The logic of an application often requires certain checks and actions when the user enters new data into an input field, double clicks an entry in a list widget, or triggers another input event. Validation and Notification properties available for certain widgets provide this facility and we will now show a simple example of their use.

Example: Validating user input

Problem: Modify Example 1 from Section A.1.4 - entering text into an input field - to prevent the user from entering and accepting digits.

Solution: This problem is a pure extension of Example 1 and we will implement it as a subclass of its solution. Before we do, however, let's explore the validation features available in VisualWorks.

For several widgets, including input field, the Properties Tool contains two special pages of additional properties called Validation and Notification (Figure A.1.17). According to User's Guide, 'validation properties are specified when you want a widget to ask its application model for permission to proceed with certain actions, namely, accepting focus, changing internal state, or giving up focus.' Validation message must return true for 'proceed' and false for 'ignore'. Notification, on the other hand, is used to inform the application model that one of the three events listed above has occurred. The message specified as a notification property is automatically sent to the application model when the event occurs.

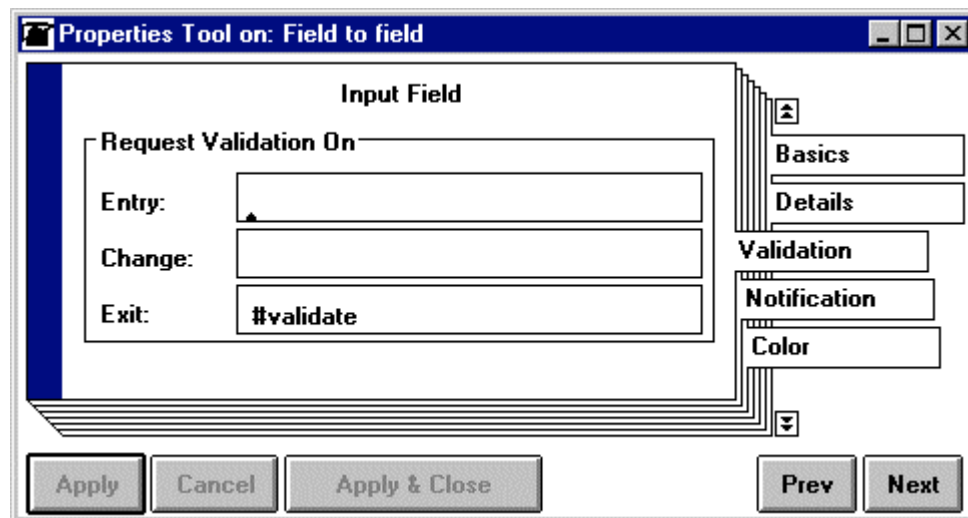


Figure A.1.17. Validation properties of input fields.

As an example, if the method specified as the widget's Exit property returns true, the new text is accepted but if it returns false, the new value is not accepted. In our example, we specified validate as the name of the Exit method. If we defined validate as

validate

"Accept new value under all conditions."
^true

the new value would always be accepted. If we defined validate as

validate

"Ignore new value under all conditions."
^false

the new value would always be ignored. In our problem, we are to check the input when the user attempts to *accept* it, and accept it only if all the characters are letters. To do this, *validate* must obtain the text entered (and displayed) in the input field and test whether all its characters are letters.

To obtain the text displayed in an input field, we cannot go to the *Aspect* value holder of the widget because the text has not yet been accepted. Instead, we must get the widget from the builder and extract the text directly from it. To be able to access the widget, we must give it an ID (we called it *#inputField*) and ask the builder for this component. The rest, by trial and error if you don't know any better, is to ask the component for the widget ('component' is the wrapper around it), get the widget's controller, get the view which holds and displays the text, and then get the contents. Altogether, we can get the string *displayed* in the input field (whether accepted or not) by sending

(builder componentAt: #inputField) widget controller view displayContents string

After getting the string, we can check whether the string contains a digit. If the answer is *true*, we want to accept the string and proceed. If the answer is *false*, we want to ignore the input. The following implementation uses this principle:

validate

"Accept the input if it consists of letters only."

| string |

string := (builder componentAt: #inputField) widget controller view displayContents string.

^string contains: [:char | char isDigit] not

This is only an introduction to validation features available in VisualWorks. We leave it to you to explore additional details of Validation and Notification in exercises.

Main lessons learned:

- Several VisualWorks widgets, including input field, provide validation of user input.

Exercises

1. Is it possible to implement our example without validation?
2. Write an application to test the exact behavior of Notification and Validation. The user interface will consist of one input field and one check box, each with all Validation and Notification properties. Each of the Validation methods will open a confirmation dialog informing the user about the type of event that occurred and allowing him or her to return *true* or *false* as the answer. Notification methods will simply open an appropriate 'warning'.

A.1.8 A Course Evaluation program

As an illustration of the use of radio buttons, we will now design and implement another simple application whose purpose is to computerize course evaluation forms.

Specification: The user opens the application by executing a Smalltalk expression containing the name of the course. When the application opens, the user can complete the form by clicking radio buttons associated with individual questions (Figure A.1.18). Exactly one answer must be selected for each question and the initial selections when the window opens are *N/A* (for 'not applicable'). The student can also add an optional comment at the bottom of the form. When finished, the student clicks the *Print and close* button and a summary of the data on the form is printed in the Transcript.

Course Evaluation Form

This is an anonymous evaluation of course

For each question, select a rating between 0 (worst) and 5 (best)

	0	1	2	3	4	5	N/A
1. Instructor is well prepared	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
2. Instructor is enthusiastic	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
3. Instructor knows the subject well	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
4. Instructor can answer questions well	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
5. Book is useful	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
6. Assignments stimulate learning	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
7. Assignment marks are fair	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
8. Time needed for assignments is reasonable	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
9. Labs are interesting (if applicable)	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
10. Course is stimulating	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
11. Overall opinion of course	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
12. Would recommend course to others	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
13. Course stimulated my interest in this area	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Use the space below to enter comments for the instructor:

Figure A.1.18. Computerized Course Evaluation form.

Using radio buttons in this problem may seem a bit unusual because we said that radio buttons should be used only when there are only a few choices and this seems hardly the case here. In fact, this guideline is still satisfied because we have a relatively small number of buttons for each question and the number of choices is thus always limited.

Design: As it is, the form simply collects information and does not perform any processing. Consequently, we will implement it as a single application model class called `CourseEvalForm`. The full list of its responsibilities is as follows:

- Allow the user to specify the name of the course when opening the application.
- Provide user interface for entering data.
- When the user clicks *Print and close*, close the window, gather the information into a suitable form, and print it in the Transcript window.

These responsibilities will be implemented by an opening method (specifying course number as an argument), an initialization method (to set up initial states of radio buttons), a set of *Aspect* methods (one for each row of radio buttons), an *Aspect* method for the comment, and an *Action* method for the *Print and close* button. We will gather the answers to the 13 questions in a 13-element array, and define an instance variable to hold the comment.

Implementation: The painting of the user interface and defining the properties is routine but we saved some time by creating one row of buttons and copying it for each additional row. The *Aspect* of all buttons in the first row is identical - *#quest1*, the *Aspect* of all buttons in the second row will be *#quest2*, and so on.

For the *Select* property of individual buttons we chose *#0* for the *Select* value of the leftmost button because this radio button corresponds to choice 0, *#1* for the next button, and so on. The *Select* value of the *N/A* button in the last column will be *#n/a*. All buttons in one column will have the same *Select* value but this is acceptable: Although *Select* must be different for each button in a *group*, it may be repeated in *different* groups. The *Aspect* of the comment text field will be called *comment*. The *Aspect* variable of the *course name* read-only field at the top of the form will be *courseNumber*.

After painting the interface, installing it, and using *Define* to define the instance variables and their associated methods, we can now define the remaining instance variables and implement the methods. The 13-element array for answers will be called *answers*. All accessing methods of the radio buttons have already been created by *Define* and we don't need to change them. For the class method that opens the application, we cannot use the *open* method because the user must be able to specify the course name. We will thus create a specialized opening method called *openOnCourse:* *aString* with course name as its argument, to be used as in

CourseEvalForm openOnCourse: 'COMP 1003'

This method must create an instance of *CourseEvalForm*, set its *courseNumber* to the value of the argument, and open the interface. The definition is as follows:

openOnCourse: aString

```
"Assign aString as course name and open Course Evaluation form."  
| form |  
form := self new.           "Create an instance of CourseEvalForm."  
form courseNumber: aString. "Assign its course number."  
self openOn: form           "Open, using an instance opening message."
```

The accessing method *courseNumber:* assigns new value to instance variable *courseNumber*, and the last line introduces a new built-in opening message called *openOn:*. The receiver of this method is an *instance of the application model* (*openOn:* is thus an *instance* method), and the method assumes that the specification of the window is stored in the default class method *windowSpec*. There are several other opening messages and we urge you to explore them. The new method is inherited from *ApplicationModel* and it sends the hook message *initialize* and all the rest as usual. We will use *initialize* to define the initial setting of the radio buttons to 'n/a' in the way explained earlier:

initialize

```
"Initialize aspect value holders of all rows to n/a for initial display"  
| symbols |  
symbols := (#answer1 #answer2 #answer3 #answer4 #answer5 #answer6 #answer7 #answer8  
            #answer9 #answer10 #answer11 #answer12 #answer13).  
symbols do: [:answer| (self perform: answer) value: 'n/a' asValue]
```

Some might consider this solution somewhat unsatisfactory because all elements of array *symbols* are essentially the same and we should not have to type them one after another. As an alternative, we can construct the symbols from strings, using the shared part *answer* and appending the index as follows:

initialize

```
"Initialize aspect value holders of all rows to n/a for initial display"  
1 to: 13 do:  
    [:index| (self perform: ('answer', index printString) asSymbol) value: 'n/a' asValue]
```

The only other method that we must write is for handling the *Print and close* button. Its principle is as follows:

printAndClose

```
"Gather answers, close window, and print results in Transcript."  
  "Collect all answers, the comment, and the course number."  
  "Close the window."  
  "Print the result in Transcript."
```

Collecting the answers will probably require many messages and we will thus define a separate message `gatherAnswers` to do this. The definition is now as follows:

printAndClose

```
"Gather answers, close window, and print results in Transcript."  
  self gatherAnswers.  
  "Close the window."  
  self closeRequest.  
  "Print results in the Transcript"  
  Transcript clear; show: 'Evaluation of course ', courseNumber value; cr; cr.  
  1 to: answers size do: [:index | Transcript show: 'question ', index printString, ' '; tab;  
    show: (answers at: index); cr].  
  Transcript show: comment value
```

To finish the program, we need the definition of `gatherAnswers` which produces the array of answers. The solution can be described as follows:

- For Question 1, find the value of instance variable `answer1` and insert it into the first element of array `answers`.
- For Question 2, find the value of instance variable `answer2` and insert it into the first element of array `answers`.
- Similar for questions 3 to 13.

To get the answer for Question 1 and to insert it into the array, we need

```
answers at: 1 put: answer1 value    "Returns, for example 3."
```

For Question 2, the code is

```
answers at: 2 put: answer2 value
```

and so on. Using this style, the definition would become

gatherAnswers

```
"Calculates array of answers"  
| array |  
  "Create uninitialized array with 14 elements: 13 for questions, 1 for comment"  
  array := Array new: 14.  
  "Calculate values of its elements"  
  answers at: 1 put: answer1 value.  
  answers at: 2 put: answer2 value.  
  answers at: 3 put: answer3 value.  
  etc.  
  answers at: 13 put: answer13 value  
  questions at: 14 put: comment value
```

Since each question is handled by an almost identical statement, we can again use the `perform:` message on a dynamically constructed name of the accessing method:

gatherAnswers

```
"Calculates array of answers"
```

```
answers := Array new: 14.  
1 to: 13 do: [:row |  
    | value |  
    value := (self perform: ('answer' , row printString) asSymbol) value.  
    answers at: row put: value asString].  
questions at: 14 put: comment value
```

We leave it to you to fill in the missing pieces and test that the application works.

Main lessons learned:

- In addition to open, ApplicationModel provides several other application opening methods such as openOn:. Some of them are class methods, others are instance methods, but all send hook methods.

Exercises

1. Implement the example from this section.
2. Check how the openOn: method in ApplicationModel works.
3. Study and describe the interface opening protocols on the class and the instance sides of ApplicationModel.

A.1.9 A (very) simple computerized Tax Form – controlling window closure

In this section, we will implement a simple tax form application with the interface in Figure A.1.19. The input fields at the top are for entering textual information. The fields below are divided into an income column on the left, a deductions column in the middle, and an information column with short help text and various action buttons on the right. The *Married Deductions* field is read-only and its contents are calculated by the program from information about marital status provided by the user. Fields *Taxable Income* and *Total Tax* at the bottom are also read-only and calculated by the program.

The function of the action buttons on the right is obvious but the *Quit* button is a bit more complex. When the user clicks *Quit*, the program first checks whether at least the first and last names, the street, the city, and the postal code have been entered. If not, it opens a dialog asking the user whether he or she really wants to quit. If the user confirms, the program closes the window and quits, otherwise the request to quit is ignored. The method also checks whether the form has been printed or saved in a file. If not, it notifies the user and closes only if the user confirms his or her desire to quit. We leave the file saving operation as an exercise (see Chapter 10 for background material).

Our tax form is a patented improvement on regular government tax forms: Unlike ordinary tax forms, we allow the user to change the tax rates and the tax brackets (the values of taxable income at which the tax rate changes) and thus decide how much money the government deserves. Three tax brackets and corresponding tax rates will be built in as default values.

Simplified Tax Return

First Name: Last Name: ☐ Married ☐ Widowed
☒ Single ☐ Separated

Street: City: Postal code:

INCOME	DEDUCTIONS
Employment: <input type="text" value="\$0.00"/>	Pension Plan: <input type="text" value="\$0.00"/>
Business: <input type="text" value="\$0.00"/>	Married Deduction: <input type="text" value="\$0.00"/>
Old Age Pension: <input type="text" value="\$0.00"/>	Union/professional dues: <input type="text" value="\$0.00"/>
Disability Benefits: <input type="text" value="\$0.00"/>	Moving Expenses: <input type="text" value="\$0.00"/>
Rental Income: <input type="text" value="\$0.00"/>	Child Care: <input type="text" value="\$0.00"/>
Farming Income: <input type="text" value="\$0.00"/>	Business loss: <input type="text" value="\$0.00"/>
Total Income : <input type="text" value="\$0.00"/>	Total Deductions : <input type="text" value="\$0.00"/>
Taxable Income = Total Income - Total Deductions : <input type="text" value="\$0.00"/>	
Total Tax from Tables : <input type="text" value="\$0.00"/>	

Instructions

White space - your input

Pink space - automatic

Commands

Print

Change tax rates

Change tax brackets

Load from File

Save to File

Help

Quit

Figure A.1.19. User interface of a simple computerized tax return form.

Design. The problem as stated is again so simple that we don't need a domain model - all data will be kept by the application model which will also do all the processing. If this was a part of a larger application, we would create a domain class to hold the tax information and we leave this extension as an exercise.

As we already know, we will need value holders for *Aspects* of all text fields. We will need some additional variables too: First, variables to hold the limits of tax brackets and the corresponding tax rates. We will call them *limit1*, *limit2*, *limit3*, and *rate1*, *rate2*, and *rate3* respectively. We also need variables to hold information about whether the data has been printed and whether it has been saved, and we will call them *saved* and *printed*. Finally, we will add a variable to keep track of whether the data has changed since the last *save* operation; this variable will be used to ignore *save* requests if there is no new information to be saved. The variable will be called *infoChanged*.

Which behaviors do we need? We must *initialize* value holders so that the window opens with zeros in all numerical fields, and we need *change* methods to change totals and taxes when the user enters new values into numeric fields. We also need change methods to notify *Married Deductions* when the user clicks marital status. Setting these things up will require a *postBuildWith:*. Finally, we need *Action* methods for action buttons.

Implementation. After painting the user interface and installing it, we specify the properties of all widgets and *Define* the necessary instance variables and stubs of *Aspect* and instance methods. We chose the following properties:

- For input fields intended for monetary values, we choose *Type FixedPoint(2)*.
- For input fields used for input of text we choose *Type String*.
- We chose names of *Actions*, *Aspects*, and *Select* parameters to match the function of the corresponding widgets (for example, *#business*, *#city*, *#single*) to minimize confusion.
- We chose background colors to indicate which fields are to be filled by the user (white) and which fields are calculated by the program (pink).

The requirements on initialization are similar to those in the Restaurant Menu because most value holders must send a message when their value changes. Note that all input fields send the same message to recalculate the totals and make sure that they are properly displayed. Finally, some variables, such as tax bracket limits, must be initialized to default values:

postBuildWith: aBuilder

| symbols |

```
"Define change messages by taking advantage of accessing methods."
symbols := #(#business #city #disability etc. ).
symbols do:
    [:aspectSymbol | (self perform: aspectSymbol)
                    onChangeSend: #newTotals to: self].
"Initialize Aspect value holder for marital status radio buttons."
maritalStatus value: #single.
"Initialize tax bracket limits and rates."
"Change of values must cause recalculation and redisplay of totals - use value."
(limit1 := 5000 asValue) onChangeSend: #newTotals to: self.
(limit2 := 20000 asValue) onChangeSend: #newTotals to: self.
(limit3 := 40000 asValue) onChangeSend: #newTotals to: self.
(rate1 := 0.1 asValue) onChangeSend: #newTotals to: self.
(rate2 := 0.2 asValue) onChangeSend: #newTotals to: self.
(rate3 := 0.3 asValue) onChangeSend: #newTotals to: self.
"We have not changed, saved or printed the information yet."
infoChanged := false.
printed := false.
saved := false
```

Note that we stored tax bracket limits and rate values in value holders even though they are not attached to any widgets. This means that we can send them *onChangeSend:to:* messages and that change of their values will automatically cause recalculation of totals and redisplay. This is a common and very useful application of value holders.

Our next task is to write the definition of *newTotals*. The underlying algorithm is a simple implementation of tax rules:

1. Calculate total income.
2. Calculate deductions, taking into account marital status.
3. Calculate taxable income as difference between income and deductions.
4. Calculate part of taxable income falling into each bracket, and the corresponding tax.
5. Calculate total tax.
6. Assign new values by value: to propagate the change to widgets.

The definition based on this algorithm is as follows:

newTotals

```
"Something has changed, recalculate and redisplay totals."  
| bracket1 bracket2 bracket3 taxable |  
"Calculate total income."  
totalIncome value: business value + disability value + employment value + farming value +  
pension value + rental value.  
"Calculate married deduction."  
marriedDeduction value: (maritalStatus value == #married  
ifTrue: [500]  
ifFalse: [0]).  
"Calculate total deductions."  
totalDeductions value: pensionPlan value + unionDues value + businessLoss value +  
childCare value + moving value + marriedDeduction value.  
"Calculate taxable income – must not be negative."  
taxableIncome value: ((totalIncome value - totalDeductions value) max: 0).  
taxable := taxableIncome value.  
"Calculate tax for each bracket."  
bracket1 := taxable > limit1 value  
ifTrue: [taxable - limit1 value min: limit2 value - limit1 value]  
ifFalse: [0].  
bracket2 := taxable > limit2 value  
ifTrue: [taxable - limit2 value min: limit3 value - limit2 value]  
ifFalse: [0].  
bracket3 := taxable > limit3 value  
ifTrue: [taxable - limit3 value]  
ifFalse: [0].  
"Calculate total tax."  
totalTax value: (bracket1 * rate1 value) + (bracket2 * rate2 value) + (bracket3 * rate3 value).  
"Reset state variables."  
infoChanged := true.  
printed := false.  
saved := false
```

We now come to *Action* methods. The *Load* and *Save* buttons are left as an exercise. The *tax brackets* and *tax rates* buttons open a dialog window and offer initial answers, *Help* opens a help window with information about the form, and *Print* constructs a string containing all information entered by the user and sends it to the Transcript. We leave these methods as an exercise. Finally, we develop the quit method which has the following previously explained tasks:

1. Check whether all required information has been entered.
2. If yes, check whether final information has been printed or saved.
3. If the last condition is satisfied, close the window and terminate the application.
4. In all other cases, a condition has been violated; ask the user to confirm that he or she wants to quit.

The basis of the quit method is the principle that when a window is asked to close, it automatically sends `changeRequest` to its application model. The window then closes if `changeRequest` returns true but the close request is ignored if `changeRequest` returns false. The default definition of `changeRequest` in `ApplicationModel` simply returns true but we can override it with our own `changeRequest` method. By applying these principles, we develop the following definitions:

quit

```
"Close the window if everything is OK."  
self closeRequest
```

and

changeRequest

```
"Close the window if everything is OK."  
| isIncomplete |
```

```
super changeRequest ifFalse: [^false].           "In case a superclass has a handler."
"Is information complete?"
isIncomplete := city value isEmpty or:
    [firstName value isEmpty or:
    [lastName value isEmpty or:
    [postalCode value isEmpty or: [street value isEmpty]]]].
"If incomplete, ask user to confirm and exit if so specified."
isIncomplete
    ifTrue: [^Dialog confirm: 'The form is not complete. Close anyway?'].
    "Has information been printed or saved? Ask for confirmation to close if not."
    (printed or: [saved])
    ifFalse: [^Dialog confirm: 'The form has not been printed or saved. Close anyway?'].
    "If we got here, everything is OK or the user confirmed so confirm that window should be closed."
    ^true
```

Note that before doing our own processing, we first executed the inherited `changeRequest` mechanism in case that a superclass provides its own handling of window closure. If this message returns `false`, the inherited mechanism requires that the window remains open and we thus return `false` and exit. Otherwise, we continue with our own tests. In our case, we could have skipped this precaution because we know that the only relevant superclass of `TaxForm` is `ApplicationModel` and its `changeRequest` simply returns `true`. However, executing the behavior defined by superclasses is a good habit and besides, what if somebody later inserted a class between `TaxForm` and `ApplicationModel` and this new superclass had its `changeRequest`?

Main lessons learned:

- The request to close a window sends `changeRequest` to the application. Its default definition in `ApplicationModel` returns `true` to close the window. This default behavior can be overridden by redeclaring `changeRequest` in the application model to provide any tests and communication with the user that may be necessary. The method must return `true` or `false`.
- At the beginning of a new definition of a method defined in a superclass, always consider whether the superclass behavior should be executed first.
- The use of `ValueHolder` is not restricted to widgets. Value holders are useful whenever we want to broadcast every change of the value of an object to other objects.

Exercises

1. Implement and test the tax form application.
2. Reimplement the tax form using a domain model and removing all tax processing from the application model.
3. We have not paid much attention to instance variables `saved`, `printed`, and `infoChanged`. Are they all necessary?
4. We used `postBuild:with:` to perform initialization. Could we have done it in the `initialize` method?
5. Modify the tax application to create a `TaxReturn` object and save it in a file. Add *Load* option.
6. Examine and describe the detailed mechanism of window closure, focusing on `changeRequest`.

Conclusion

In this appendix, we presented radio buttons and check boxes, explained the use of the input field widget, and illustrated these new widgets on several examples. We also explained some of the inner working of the UI builder.

All programmers involved in the design of user interfaces must remember that interface design has two aspects: The technical aspect of declaring the widgets' functionality, and adherence to GUI conventions

and design rules. Ignoring established conventions may make the user interface difficult to use. Conventional uses of buttons and check boxes are as follows: Radio buttons are used to select one of a limited number of known alternatives. When the number of choices is large or when it is not fixed or known beforehand, use single-selection list widgets instead. Check boxes are used individually or in groups of independent boxes to select any number of choices when the number of choices is not large and when the choices are known beforehand. When the number of choices is large or when it is not fixed or known beforehand, use multiple-selection list widgets instead.

The technical aspects of the use of check boxes and radio buttons are as follows: A check box is used for selecting or deselecting a choice. It requires the specification of an *Aspect*, the name of an instance variable holding a Boolean *ValueHolder* representing the state of the box (true for on, false for off). It is also the name of the method accessing this variable. The variable and the method can be automatically defined with *Define* and do not require any further modification if the box is initially off.

Groups of mutually dependent radio buttons achieve their dependence by sharing one *Aspect* - a value holder with the current selection. A radio button has two essential properties: *Selection*, and *Aspect*. The *Selection* is a *Symbol* which is assigned to the shared *Aspect* of the group when the button is clicked on. Each button in a group has its own *Selection* value. Use the *initialize* method to assign a *ValueHolder* with the *Selection* value of the button that should be on when the window first opens to the *Aspect* variable.

An input field is a one-line text editor. Its only essential parameter is *Aspect*, the name of an instance variable holding a *ValueHolder* on the text accepted by the user, and the corresponding accessing method. If the text field is to display text when it opens, a *ValueHolder* with this text must be assigned to the *Aspect* variable in the *initialize* method.

To store the text displayed in an input field in the *Aspect* variable, the user must accept it by pressing <Enter> or by executing *accept* from the field's <operate> menu. Until the user accepts the text, the accepted value may be different from the value displayed and it is thus held in a separate instance variable of the input field.

To force a variable holding a *ValueHolder* to execute a specified message whenever its value changes (via *value:*), send *onChangeSend:to:* to the variable in the *initialize* method. However, don't forget to assign a *ValueHolder* to the variable first. Alternatively, you can send the *onChangeSend:to:* message to the value holder directly via the builder or via its accessing method. The message given as the first argument of *onChangeSend:to:* will be sent whenever the receiver gets a *value:* message. The second argument is normally *self* because the change message is usually defined in the application model. The *onChangeSend:to:* mechanism is the basis for setting up links between widgets.

Each GUI widget may have an ID. IDs of widgets accessed by the same builder must be unique because the builder keeps them in a registry. A widget needs an ID only when we must access it at run time, for example to enable, disable, show, or hide it. Sending *componentAt: anID* to the builder returns the widget's wrapper. The builder itself can be obtained from the application model.

In addition to the *open* class method for starting an application, *ApplicationModel* contains several other opening methods, both in its instance and in its class protocols. All provide access to the hook methods explained earlier.

In addition to hooks into the window opening process, Smalltalk also provides a hook into the window closing process. To close a window programmatically, send *closeRequest* to the application; this, in turn, sends *changeRequest*. The default definition of *changeRequest* in *ApplicationModel* returns true but this default behavior can be overridden by redefining *changeRequest* in the application model to provide any necessary tests and communication with the user. If the *changeRequest* method returns false, the request to close is ignored. When redefining *closeRequest* and other methods defined in a superclass, consider executing the behavior defined in the superclass first.

Code can sometimes be simplified by constructing messages at run time. The *perform:* message may then be useful.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

DependencyTransformer.

Widgets introduced in this chapter

check box - used to turn a feature on or off. *Aspect* is a ValueHolder on a Boolean

input field - one-line text editor. *Aspect* is a ValueHolder of accepted text. User must press <Enter> or use *accept* to store the text in the *Aspect* variable

radio button - used in groups to select one of several mutually exclusive choices. Requires *Aspect* and *Selection* properties. *Aspect* is ValueHolder containin *Selection* symbol, shared by all buttons in a group, *Selection* is a symbolic value of *Aspect* for a given button.

Terms introduced in this chapter

change notification - specifying that a change in a ValueHolder should send a message, usually to the application

check box - square button used for turning a selection on or off

input field - one-line text editor

radio button - round button used in groups to select one of several mutually exclusive alternatives

registry of named components – dictionary associating IDs of UI component and their wrappers; held by the application builder

Selection property - Symbol associated with a radio button; assigned to the shared *Aspect* variable when the button is clicked on

widget ID - optional unique ID assigned to a widget so that the application can communicate with it at run time

Appendix 2 - Dataset, Subcanvas, Notebook, Dialog Window, Menus

Overview

Dataset widgets look like tables but all their rows have the same structure and display items of the same kind. Individual cells may be check boxes, combo boxes, or input fields.

A subcanvas is a rectangular window area holding another canvas. The contents of a subcanvas area may be changed at run time. One of the main uses of subcanvases is in notebook widgets because each notebook page is a subcanvas.

Notebooks are familiar from the UI Properties Tool. Navigation through notebook pages is via one or two sets of tabs. One set of tags called major tags is required, minor tabs are optional.

A dialog window, also called a modal or pre-emptive window is a window that retains focus (does not let the user activate any other window) until the dialog is closed.

Menus that can be created with the UI Painter include popup menus, menu buttons, and menu bars. Popup menus are the familiar menus that open when the <operate> button is pressed, menu bars are the drop down menus residing on the top of the window, and menu bars are drop down menus that look like input fields.

A.2.1 Dataset widgets

A dataset looks like a table (Figure A.2.1) but its purpose and behavior are somewhat different. Its main characteristics are as follows:

- Datasets are used to display a list of related multi-component objects, all instances of the same class. (Tables, which are in many ways similar to datasets, can display heterogeneous information.) Two examples of the use of dataset are displaying a list of employees and displaying a list of books in a library catalog.
- All items in the same column of a dataset are accessed by the same method. As an example, items in the *Author* column in Figure A.2.1 might all be accessed by method *author*. Individual cells are accessed by Point coordinates.
- Cells may display text, combo boxes, or check boxes. (Tables display only non-editable text.)
- Dataset cells are directly editable 'in place'.
- The data model of a dataset is a *SelectionInList* object whose List components are usually arrays of objects displayed in row cells. As an example, each row in the dataset in Figure A.2.1 is an array whose elements are strings and numbers representing components of Book objects including author name, book title, and year of publication. (The model of a table is a *SelectionInTable*.)
- Datasets have simpler decorations.

We will now demonstrate datasets on two examples.

Example 1: Dataset as a user interface to a book catalog

Problem: Implement the user interface for accessing and editing a catalog of Book objects shown in Figure A.2.1. Each Book has an author, a title, and a year of publication; author and title objects are strings, year is an integer. All cells in each row must be editable. The interface will be used in conjunction with data stored in a file but for this test, implement it so that it will open with a few Book objects already in place; additional books can be entered by the user at run time.

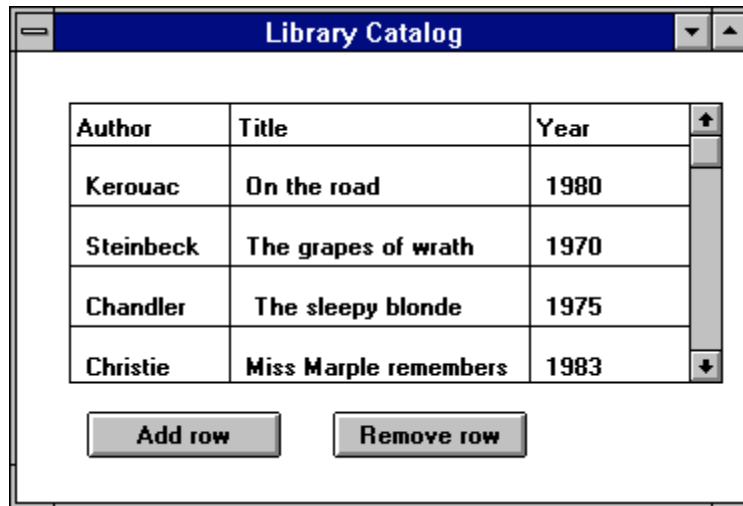


Figure A.2.1. Desired library catalog interface.

Solution: Two classes are involved in the solution: Book (simple subclass of Object holding book information - left as an exercise), and an application model (we will call it LibraryCatalog). To implement the solution, define Book, paint the interface, install it on an application model, define widget properties, and define the necessary methods. The procedure for specifying a dataset is a bit more complicated and must be followed carefully:

1. After painting the dataset widget, specify its *Aspect* property, the name that will return a SelectionInList object. We will call it bookList (Figure A.2.2).
2. To create the three columns required by the specification, click *New Column* in the *Properties* tool (Figure A.2.2) three times.
3. Specify the properties of each column. To do this, *select* the column by holding the <Ctrl> key down and clicking the <select> button with the mouse cursor inside the column. The column will be highlighted.
4. Select the *Column* page of the Properties Tool (Figure A.2.3) and fill in the following information¹:
 - i. Column label (*String* property) - we chose *Author* for the first column.
 - ii. Aspect expression. This expression will return the data for the cell at the intersection of this column and the row currently selected by the user. Our expression is

selectedBook author

where selectedBook is the name of an instance variable holding a Book object (defined in our application model to hold the selected row); author is the name of the accessing method that extracts the author value from the Book object. This method is defined in class Book.

- iii. Change the width of the column if desired.
 - iv. Specify the desired *Type* of cell - the available choices are shown in Figure A.2.3 and we selected *Input Field* for each column because we want to be able to edit the cells at run time.
Repeat this step for each column (with expression selectedBook title and selectedBook year). The third column is slightly different because its value year is a number and requires *Number* as *Data Type* in the *Column Type* page of Properties. With this type, the widget will automatically convert between the string-based display and the number-based year object.
5. *Install* the canvas and *Define* all properties. Examine the definition of bookList in the aspect protocol to see its non-trivial character.

¹ If the *Column* page is disabled, you have not selected the column in the Canvas.

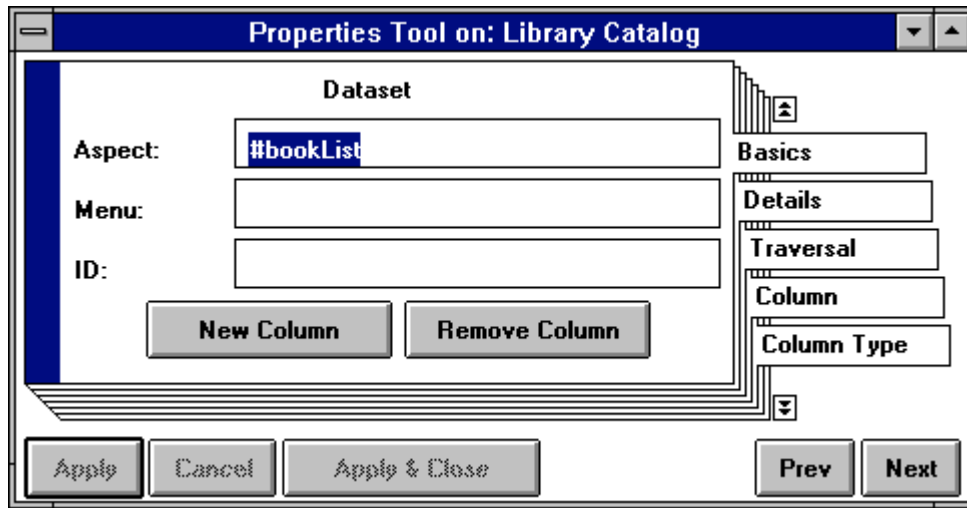


Figure A.2.2. The essential part of the Basic page of the Properties Tool.

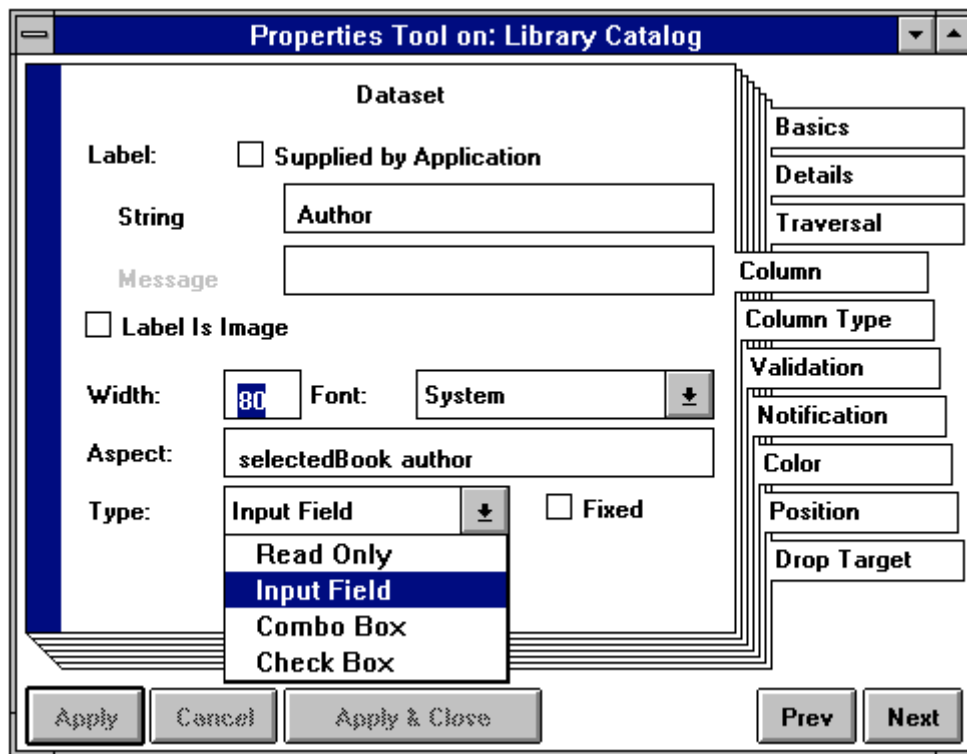


Figure A.2.3. The main part of Column Properties.

Methods: We will start with the initialize method for LibraryCatalog. It will create the underlying SelectionInList model for the dataset, and initialize its list component. The algorithm is as follows:

1. Create a List whose elements are the rows of the dataset that will be displayed when the window opens. If the window were to open with an empty dataset, we would not put any elements into the List.
2. Create the SelectionInList object for the dataset *Aspect* by sending the *Aspect* message bookList to the model . Then send the list: message with the list created in Step 1.

The whole definition is as follows:

initialize

```
"Create and initialize the SelectionInList underlying the dataset widget."  
| list authors titles years |  
"Create a List object to hold the data."  
list := List new.  
"Initialize the list - we use three arrays representing columns to do this more compactly."  
authors := #('Kerouac' 'Steinbeck' 'Chandler' 'Christie' 'Homer' 'Kundera').  
titles := #('On the road' 'The grapes of wrath' 'The sleepy blonde' 'Miss Marple remembers' 'Ilias'  
            'Farewell party').  
years := #(1980 1970 1975 1983 11 1980).  
"Create the rows."  
1 to: authors size do: [:index | list add: (Book author: (authors at: index)  
                                                title: (titles at: index)  
                                                year: (years at: index))].  
"Create the SelectionInList model of the dataset and assign the list to it."  
bookList := self bookList list: list "A somewhat malicious use of the identifier list."
```

We assume that instances of *Book* are created with the *author:title:year:* message.

The next task is to create action button methods *addBook* and *removeBook* for adding and deleting rows (Figure A.2.1). The essence of both methods are *List* messages for adding and removing elements. Since the dataset widget is the list's dependent, changes to the list automatically redraw the widget.

Adding a row

To add a row, get the *SelectionInList* model of the dataset using the *Aspect* message *bookList*, get its *List* via *list*, and add a new *Book* object with the *add:* message. This adds an empty row and the user will now presumably enter data into it. The definition is

addBook

```
"Get the List object and add a new item of the appropriate kind."  
self bookList list add: Book new
```

Removing a row

This action assumes that the user has selected a row, and checking that this is indeed so is the first task of the method. The method asks the *SelectionInList* object for the index of the selected row. If the index is 0, no row is selected and no action will be taken. Otherwise, the method sends *removeAtIndex: index* to the *List*. The definition is as follows:

removeBook

```
| index |  
index := bookList selectionIndex.  
index = 0 ifTrue: [^self].  
bookList list removeAtIndex: index
```

This completes the solution.

Example 2: Add sorting buttons to Library Catalog

Problem: As an improvement of the library catalog, add three buttons to allow the user to sort book entries by author, by title, or by year. The desired user interface is as in Figure A.2.4.

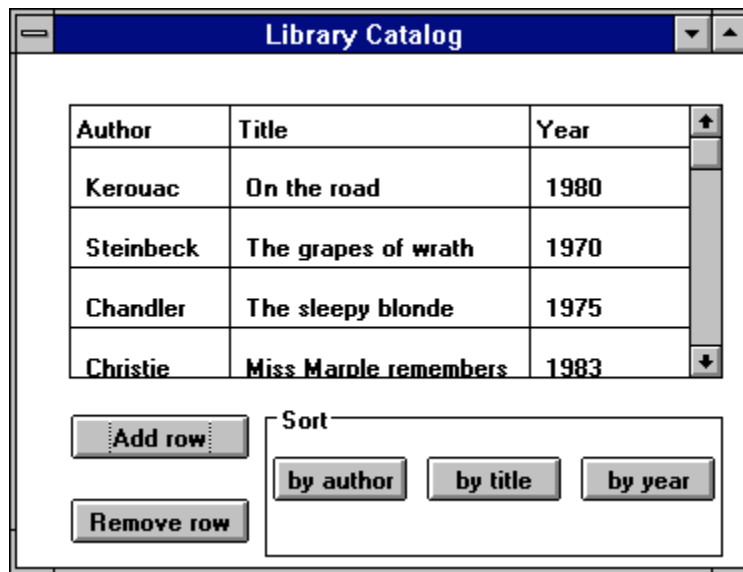


Figure A.2.4. Library catalog for Example 2.

Solution: To sort the list by author, we must get the list from the SelectionInList object, ask it to sort itself with a sort block based on book authors, and assign the result to bookList via the list: message; this takes care of redrawing the dataset via dependency. The definition is as follows:

byAuthor

"Resort the list by author names and redisplay it."

```
bookList list sortWith: [:book1 :book2 | book1 author < book2 author].  
bookList list: bookList list
```

We leave the remaining sorting buttons to you and conclude this section with a comment that may save you some frustration. In our first attempt at byAuthor, we wrote the definition as follows:

byAuthor

```
bookList list: (bookList list sortWith: [:book1 :book2 | book1 author < book2 author])
```

This did not work because the sortWith: message returns a SequenceableCollectionSorter rather than the sorted List.

Main lessons learned:

- Dataset widgets look like tables but their behavior and underlying models are different.
- A dataset is essentially a list widget whose individual lines are multi-item objects.
- The underlying dataset objects are SelectionInList whose list elements are rows, and a *selected row* object. Cells may be input fields, check boxes, or combo boxes. The nature of each cell in a given column is the same and its contents are obtained via the *Aspect* of the column.
- Row elements are usually consecutive elements of an array.
- The type property of a dataset column ensures automatic conversion between the cell and the internal representation of the displayed object. Built-in types include String, Number, Time, Boolean, and others.

- Dataset properties are more complicated than properties of most other widgets and the procedure for creating and using them must be followed carefully.

Exercises

1. Complete Example 2.
2. Add a *Print* button to the library catalog. In an initial implementation, clicking *Print* will display the catalog in the Transcript in a format similar to the dataset. In a second implementation, *Print* will print the dataset on a printer.
3. One of the data types available for columns is **Text**. Modify the library catalog to display author names in italics. (Hint: Conversion can be performed via the column *Aspect* expression.)

A.2.2 Subcanvas

Many application windows contain parts that are themselves applications (for example a digital clock). Others contain parts that are swapped with other parts at run time. And many applications contain groups of widgets that are repeated in several windows – the Property Tool is an example. Subcanvases are used in all these situations and they are also the basis of notebook widgets because every notebook page is a subcanvas.

A subcanvas is basically a rectangular window space that displays another canvas. To make it possible to display a canvas in the subcanvas area, the properties of a subcanvas include the name of the class with the specification and support for the inserted canvas, the name of the canvas specification method, and the name of the method returning the application that provides the initial contents of the subcanvas. These properties allow a variety of behaviors but in this section, we will demonstrate only the simplest one – using a subcanvas to swap a part of two alternative versions of a larger canvas back and forth. More complicated uses will be presented later.

Example: Using subcanvases to swap a part of a canvas at run time

Problem: Create a user interface whose look is controlled by a pair of radio buttons (Figure A.2.5). When the user selects *Interface 1*, the label at the bottom of the window becomes *This is Interface 1*; when the user selects *Interface 2*, the label becomes *This is Interface 2*.

Solution: We will implement the two alternative labels as swap-in canvases to illustrate the subcanvas concept. In this case, the same effect could be achieved more easily by changing the label via the builder at run time or by creating the main canvas with both sets of labels, and selectively hiding and showing them as required; but our goal is to introduce the use of a subcanvas in the simplest way possible.

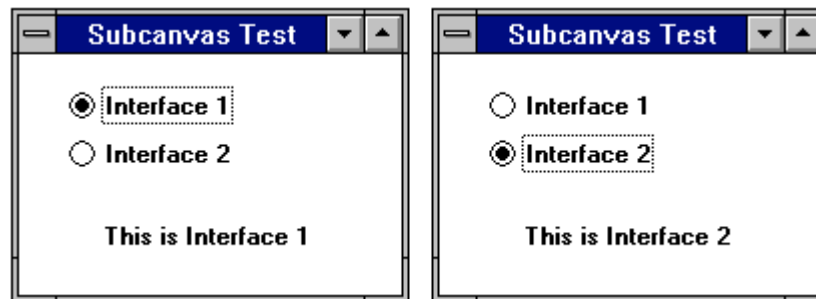


Figure A.2.5. The two alternative looks of a user interface.

To solve the problem, we will create three canvases. The first one will be the ‘main’ window with two radio buttons and a subcanvas widget for the label. The second canvas will provide the first label and the third will contain the second label (Figure A.2.6).

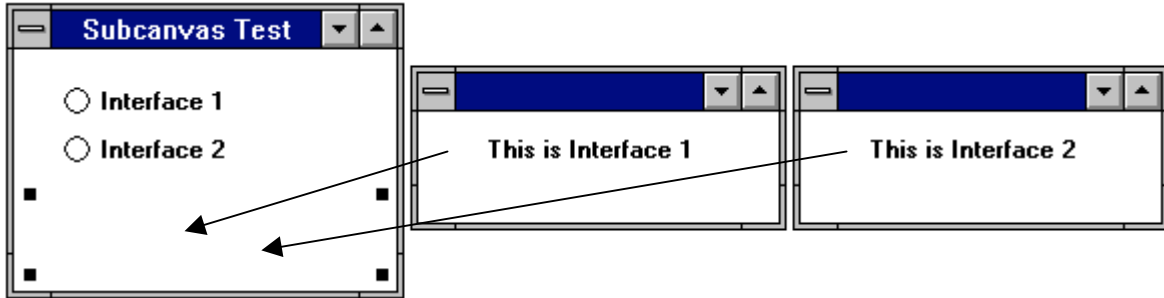


Figure A.2.6. The main canvas (left), and the two subcanvases. The subcanvas component of the main canvas is identified only by its handles because we turned its *Bordered* property off.

We start by painting the three canvases with the UI Painter. On the main *Properties* page of the *subcanvas* widget of the main canvas (Figure A.2.7), we must specify the name of the method that returns the application model of the subcanvas specification (*subcanvasAppModel*), and an ID because we will access the subcanvas via the builder to swap a new interface in when necessary.

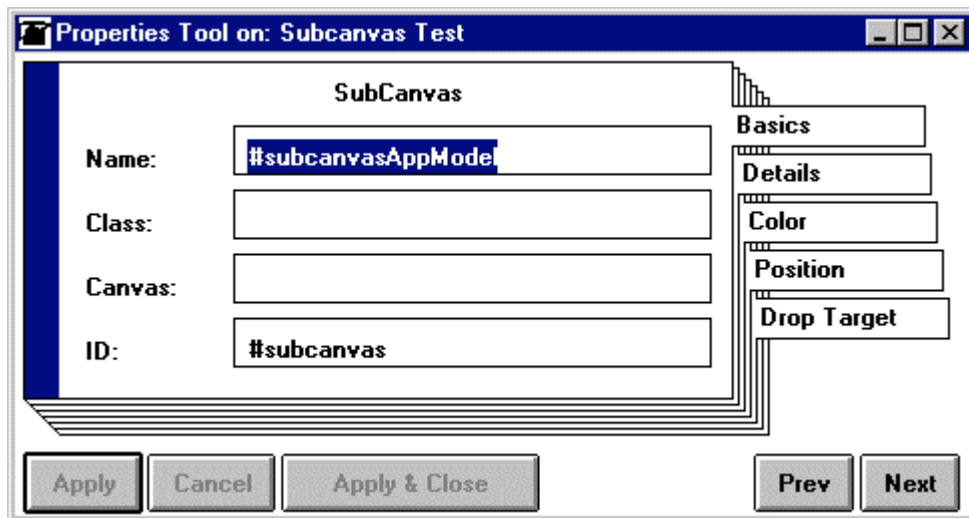


Figure A.2.7. The Basics *Properties* page of the subcanvas widget.

After creating and installing the main canvas (class *SubcanvasTest* and spec method *windowSpec*) we created the two swap-in canvases, exactly as any other canvas, and installed them on the same class with spec methods *interface1* and *interface2* respectively.

The next step is to write an *initialize* method. Its tasks will be to

- initialize the radio buttons so that the window opens with the *Interface 1* button on
- register interest in changes in radio button settings.

The definition is as follows:

initialize

```
"Activate radio button #interface1, register interest in button setting."  
interface := #interface1 asValue.  
interface onChangeSend: #newInterface to: self
```

Next, we must write the change method *newInterface* which is sent when the user clicks a radio button. The method gets the subcanvas, the spec method's symbol (*#interface1* or *#interface2*) from the

aspect variable shared by the radio buttons, and tells the subcanvas to display it. This last step is performed by the client:spec: message:

newInterface

"Switch to the subcanvas corresponding to the active radio button."

| subcanvas |

"Get subcanvas widget from the builder."

subcanvas := (self builder componentAt: #subcanvas) widget.

"Assign to it the canvas described in the appropriate spec method defined in this class."

subcanvas client: self spec: interface value

Our solution is very simple because we used the same names (#interface1 and #interface2) for both the *Selection* properties and for the names of the corresponding canvases.

The next task is to write the method that will ensure that the window opens with the correct subcanvas. The definition simply sends the newInterface message:

postOpenWith: aBuilder

"Display the initial subcanvas."

self newInterface

Note that sending message newInterface at this point is valid because we have already selected a radio button in initialize. Finally, the method that supplies the application model containing the specification of the subcanvas returns self because the subapplication is the model itself.

subcanvasAppModel

^self

The builder sends this message when it builds the bindings dictionary associating model names and their values. (Models of widgets such as input fields are their value holders, the model of a subcanvas is the application model that defines it.)

Main lessons learned:

- A subcanvas is a space holder widget that can be used to swap different user interfaces into a predetermined area at run time.
- A subcanvas is specified by a spec method that may be defined in the main application model or in other application models. This makes it possible to swap a whole application into another application.
- If the purpose of a subcanvas is limited to swapping interfaces, the same result can often be achieved by hiding and showing groups of overlaid widgets or by changing the components via the builder at run time.
- The underlying class of the subcanvas widget is SubCanvas. It provides several messages that make it possible to swap in an interface defined in an arbitrary class and window spec method.
- The model of a subcanvas is the application model that defines it.

Exercises

1. Reimplement the example from this section using
 - i. overlaid widgets and hiding
 - ii. a single label widget whose value changes at run time.
2. Trace the opening of our example application and write a description of the building process.

A.2.3 Diary - Using a subcanvas to reuse a complete application

In this section, we will build a simple diary with the user interface shown in Figure A.2.8. Clicking a day button in the calendar will display notes stored for this day in the text editor at the bottom of the window. The text can be edited and saved with the *Save notes* button. To switch to another month and year, the user clicks the *New date* button. This opens a dialog requesting the number of the month and the calendar then opens on this new date. When the diary first opens, the calendar displays the date of the day.

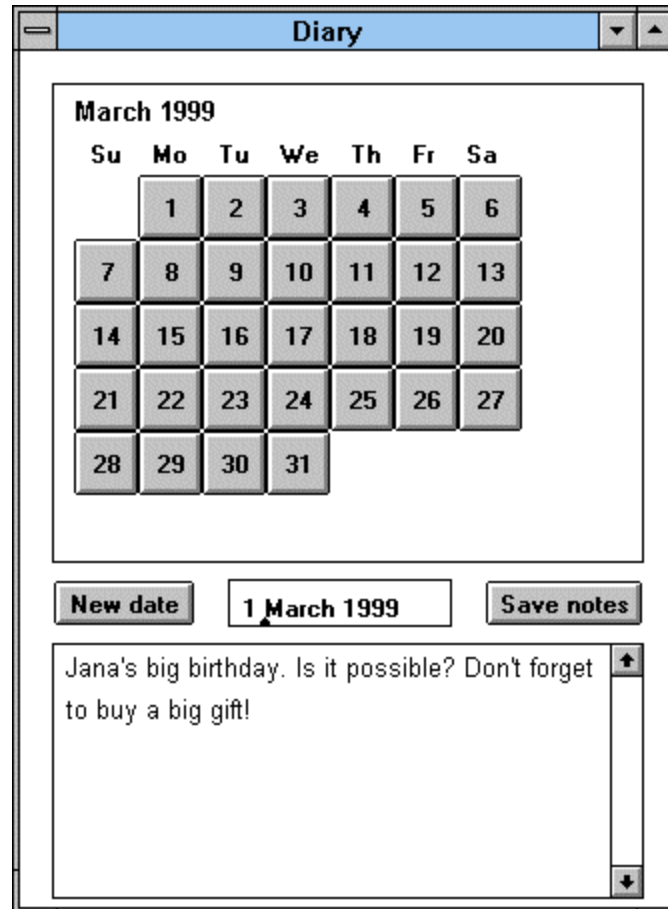


Figure A.2.8. Diary interface.

When you think about this application, you will realize that the calendar part looks like a useful component that might be reusable in other applications and we will thus create *two* application models - class *Diary* for our problem, and the reusable class *Calendar*. Class *Calendar* will be designed to be used as a new 'widget', somewhat like an extension of the palette, a stand alone application that can be plugged into a subcanvas and provide the necessary communication with its 'master' window and its application model.

Design and implementation of Calendar

Calendar is a sub-application, the user interface component in the upper half of the *Diary* window. It displays the currently selected month and year, abbreviated names of the days of the week, and several rows of action buttons with dates. The buttons are arranged to match the days of the week, and the numbers displayed on them correspond to the days in the selected month and year. The layout and contents of the canvas is given by the selected month.

One way to design the day-buttons part of the user interface is to use the UI painter to draw enough buttons to satisfy even the most demanding months (Figure A.2.9). To display a calendar for a particular

month, we would then show and enable only the required buttons and display the appropriate number as a label on each of them. We will leave this approach as an exercise and use a different strategy.

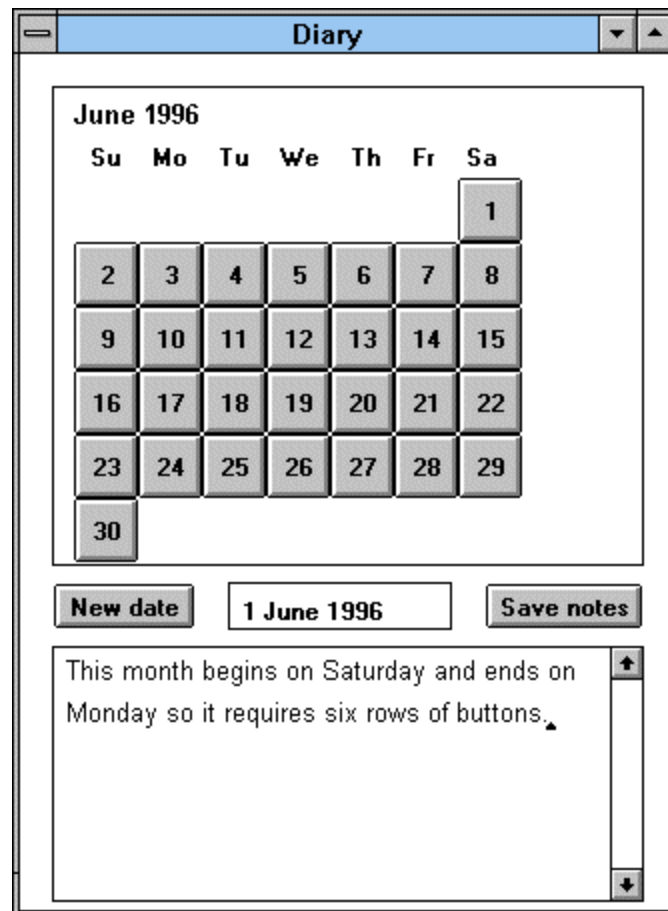


Figure A.2.9. A month that requires six rows of buttons.

The approach that we will use is to paint all the fixed parts of the calendar (button labels and names of the days of the week) *programmatically*: The builder will add the buttons and change labels to display the correct month and year at building time.

To open the calendar, the builder needs the month and the year to calculate the layout of the day buttons and their number. This information will be stored in two instance variables called *month* and *year*, both integers. In principle, the creation message should look like

```
Calendar month: 3 year: 1996
```

but since we want to be able to reuse *Calendar* as a subcanvas, in other words as a part of another application, it must be able to communicate with the application model of the main window. This means that the *Calendar* must be able to tell the window's model when the user clicks a date button. And to be able to communicate with the master application, the *Calendar* must have a reference to its application model. We will thus add an additional instance variable called *master* and the creation message will be as follows:

newOnMonth: monthInteger year: yearInteger from: appModel

```
^(self new) month: monthInteger;  
    year: yearInteger;  
    master: appModel
```

where month:, year:, and master: are simple accessing messages. The typical use of the method is as in

Calendar newOnMonth: 3 year: 1996 from: self

Our next step is to define the `postBuildWith: aBuilder` method of `Calendar`, to add day buttons to the painted background and change the labels. For adding new components programmatically (at run time), `UIBuilder` provides the `add: aComponent` method where `aComponent` is obtained in a way that depends on the component's nature. Examples in class `Builder` provide templates for several kinds of widgets. In our case, we need an action button and the expression for creating one is

`ActionButtonSpec` model: `aBlock` label: `aStringOrText` layout: `aRectangle`

where layout: is a `Rectangle` defining the upper left and lower right corners of the button (in window coordinates), and block: is the action executed when the button is clicked; this is similar to the *Action* property of an action button but arguably more powerful because it is a block rather than a message. The following is an example of a complete button-specifying expression in the context of our creation message:

```
aBuilder add: (ActionButtonSpec  
    model: [master selection: date]      "Evaluated when button is clicked."  
    label: date printString  
    layout: (x1 @ y1 extent: x2 @ y2))
```

To create a whole array of rows and columns of buttons, we will repeat the button adding statement for every button in the array. Because of the variety of possible month and day layouts, the loop will iterate over six rows and process seven buttons in each. For each button (each row-column cell) it will first calculate the corresponding date. If the date is at least 1 and not greater than the last day of the month, the builder will add the action button with the date displayed. Buttons that don't satisfy the 'within the month' condition are ignored. At the beginning of each new row, the x position (the upper left corner of the button) is reset; at the end of each row, the y position is incremented. The x coordinate is incremented after each column. The complete code to create the buttons is as follows:

```
1 to: 6 do: [:row | "Outer loop - one row at a time. Initialize x coordinate of first button."  
    x := xStart.  
    1 to: 7 do: [:column | | date | "For each row, enumerate for all 7 positions."  
        date := boxNumber - startBox. "Convert button position to date."  
        (boxNumber >= (startBox + 1) and: [date <= maxDay])  
        ifTrue: [ "We have a valid date - add the button."  
            aBuilder add: (ActionButtonSpec  
                model: [master selection:  
                    (Date newDay: date  
                        monthNumber: month  
                        year: year)]  
                label: date printString  
                layout: (x @ y extent: width @ height)).  
            "Calculate position and box number of next button in the current row."  
            x := x + width.  
            boxNumber := boxNumber + 1].  
        "End of row, update the y coordinate for the next row of buttons."  
        y := y + height]
```

where `boxNumber` starts from 1 and `startBox` is the number of the first day of the month in the week. As a result of the `model: argument`, clicking a day button sends message `selection:` with the date corresponding to the clicked button to the master object - the application containing the calendar. Any application using

Calendar will thus have to define method selector: aDate to deal with a new selection in the Calendar sub-application. In the diary application, for example, selection: will check whether a message was recorded for aDate and display it in the text view.

To complete the postBuildWith: method, we must determine how to calculate the date from month and year. To do this, we need to know the first day of the month and the number of days in a month to lay out the buttons and to number them. After examining the protocols of class Date, we find that this information can be obtained as follows:

"Construct the Date object corresponding to the first day of the month."

```
startDate := Date newDay: 1
              month: (nameOfMonth := Date nameOfMonth: month)
              year: year.
```

"Convert startDate to day name and its number within the week."

```
startBox := Date dayOfWeek: startDate weekday. "Number of start box in first row."
```

"Calculate number of days in the current month - needed to recognize last row."

```
maxDay := Date daysInMonth: nameOfMonth forYear: year.
```

With this information, we can now write the whole definition:

postBuildWith: aBuilder

"Add day buttons to the fixed part of the interface, change label to show month and year."

```
| xStart yStart width startBox boxNumber x y nameOfMonth startDate lastDay |
```

"Change date label to display current month and year."

```
startDate := Date newDay: 1 month: (nameOfMonth := Date nameOfMonth: month)
              year: year.
```

```
(aBuilder componentAt: #date)
```

```
  label: (Label with: (nameOfMonth asString , ' ', year printString) asText allBold).
```

"Calculate position of first valid button and the number of days in current month."

```
startBox := Date dayOfWeek: startDate weekday. "Number of start box in first row."
```

```
lastDay := Date daysInMonth: nameOfMonth forYear: year.
```

"Initialize upper level corner coordinates of the leftmost button in the first row."

```
xStart := 10.
```

```
yStart := 45.
```

```
y := yStart.
```

```
width := 32.
```

```
boxNumber := 1.
```

"Execute loop for 6 rows."

```
1 to: 6 do: [:row | x := xStart. "First button in the row."
```

```
  1 to: 7 do: [:column | | date | "7 columns."
```

```
    "Number of date for this position in the button array."
```

```
    date := boxNumber - startBox.
```

```
    "Display only buttons corresponding to valid dates in the month."
```

```
    (boxNumber >= (startBox + 1) and: [date <= lastDay])
```

```
      ifTrue: [aBuilder add: (ActionButtonSpec
```

```
        model: [master selection: date]
```

```
        label: date printString
```

```
        layout: (x @ y extent: width @ width))].
```

```
    "Calculate x position for next button in the row."
```

```
    x := x + width.
```

```
    "Update button number."
```

```
    boxNumber := boxNumber + 1].
```

```
  "Increment y to start of next row."
```

```
  y := y + width]
```

This method is too long and we leave it to you to split it into several more elementary methods. Add accessing methods and test, executing, for example

(Calendar onMonth: 3 year: 1996 master: self) open

A more flexible implementation

One weak point of our new 'widget' is that every application that wants to use it must implement a method called `selection`: for communication with `Calendar`. If the master application already has such a method and uses it for another purpose - for example because another sub-application also requires this name - the `Calendar` class must be redefined. This is, of course, possible, but very undesirable. What if five other sub-applications used the same selector name?

A tempting solution is to rename the method to something that is very unlikely to be used by another sub-application, such as `calendarSelection`: but this does not guarantee that the name will be unique. As a general principle, 'hardwiring' the name of a communication method of a reusable sub-application is not a good strategy.

If hardwiring is not desirable, we should make the selector 'programmable'. In other words, the master application should *tell* `Calendar` what is the name that it wants to use for communication as in

```
Calendar newOnMonth: 3 year: 1996 from: MeetingPlanner message: #calendarSelection:
```

or

```
Calendar newOnMonth: 3 year: 1996 from: self message: #selection:
```

With this approach, any application using `Calendar` is free to use any 'callback' selector it wants. The implementation of this idea is quite simple. We only need to add a new instance variable to `Calendar` to hold the name of the callback message, modify the creation message, and change the button handling message which performs the callback (in `postBuildWith:`).

The new definition of the creation message is

```
newOnMonth: monthInteger year: yearInteger from: appModel message: aSymbol
    ^(self new)      month: monthInteger;
                    year: yearInteger;
                    master: appModel;
                    message: aSymbol
```

where `message:` is an accessing method for the new instance variable `message`. The `postBuildWith:` method which specifies the message sent by clicking the date button must be modified as follows:

postBuildWith: aBuilder

```
"Add day buttons to the fixed part of the interface, change label to show month and year."
| xStart yStart width startBox boxNumber x y maxDay nameOfMonth startDate |
etc.
"Calculate position of first valid button and the number of days in current month."
boxNumber := 1.
"Execute loop for 6 rows."
1 to: 6 do: [:row | x := xStart.          "First button in the row."
    1 to: 7 do: [:column | | date | "7 columns."
        etc.
        (boxNumber >= (startBox + 1) and: [date <= lastDay])
        ifTrue: [aBuilder add: (ActionButtonSpec
            model: [master perform: message with:
                (Date newDay: date
                    month: nameOfMonth
                    year: year)]
            label: date printString
            layout: (x @ y extent: width @ width))].
        boxNumber +:= 1.
    ].
    x +:= 1.
].
etc.
```

Note an important detail: When the user clicks a day button, its action method evaluates the block

```
[master perform: message with: (Date      newDay: date
                                   month: nameOfMonth
                                   year: year)]
```

with arguments `date`, `nameOfMonth`, and `year`. Although the block is not at this point evaluated within `postBuild:` (this message was already executed), its arguments (temporary variables declared in `postBuild:`) have the values assigned to them in `postBuild:`. In other words, the block executes within the context in which it was originally defined, carrying its context with it. This is a very important property of blocks and a reason why blocks should only use their arguments and their own temporary variables if possible.

An implementation with an adapter

Another way to avoid a hardwired method is to insert a proxy, an adapter, between the two communicating objects (Figure A.2.10). This proxy translates the communication from the 'language' used by the message sender into the language of the receiver, and possibly also in the opposite direction. A class called `PluggableAdaptor` that performs this tasks for user interface widgets is in the `VisualWorks` library. This class is designed for a specific purpose in user interfaces and its general use is limited but another pluggable adapter can easily be designed along the same lines. We leave this approach as an exercise.

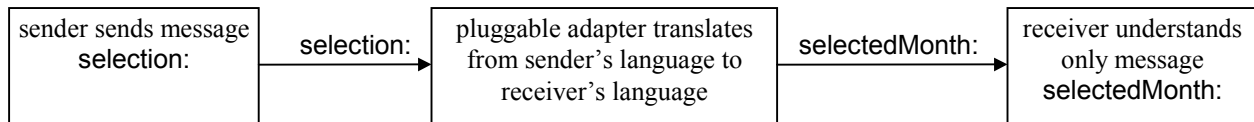


Figure A.2.10. The concept of a proxy/pluggable adapter.

Design and implementation of Diary

After creating the `Calendar` sub-application, we are now ready to implement the diary. We will implement it as class `Diary`, and its task will be to define the user interface, provide communication with `Calendar`, and hold the diary information (instance variable `notes`). The window includes the input field (aspect `selectedDate`), the buttons, the notes Text Editor (aspect `note`), and a Subcanvas with the `Calendar`. The Subcanvas properties entered via the Properties Tool will be as follows:

Property	Value	Purpose
Name:	<code>#newCalendar</code>	name of <code>Diary</code> method that supplies the new subcanvas
Class:	<code>#Calendar</code>	name of class defining sub-application
Canvas:	<code>#windowSpec</code>	name of <code>Calendar</code> method containing subcanvas specification
ID:	<code>#subcanvas</code>	to provide access the subcanvas

When you define the properties of the date input field, select type *Date* (its value holder will hold a *Date* object) and a suitable display format.

According to the specification, the `Diary` must open on today's date. If we use the standard open message, we can use the initialization method to assign the proper date to the *Aspect* property of the date text field (we called it `selectedDate`), and initialize the Text Editor:

initialize

```
selectedDate := Date today asValue.
notes := Dictionary new
```

For now, we don't assume that the notes are held in a file. We will hold the notes in a Dictionary called `notes` whose keys will be dates and values the corresponding notes, if any.

As the builder builds the canvas, it sends message `newCalendar` (see table above) to itself to obtain a `Calendar` and since the `Calendar` does not yet exist, we must now create it. Using the results of the previous section, the definition of `newCalendar` is thus as follows:

newCalendar

```
"Return and possibly calculate appropriate date by lazy initialization."  
^newCalendar isNil  
  ifTrue: [| date |  
    date := Date today.  
    ^newCalendar := Calendar  
      newOnMonth: date monthIndex  
      year: date year  
      from: self  
      message: #selection:]    "Assume callback selector selection:"  
  ifFalse: [newCalendar]
```

You can now test that opening a diary with

Diary open

gives the desired result.

When the user clicks a date button, class `Calendar` sends the `selection:` message (according to the `newCalendar` method above) which must change the date in the input field (aspect `selectedDate`) and the displayed note (aspect `note`). The definition is thus

selection: aDate

```
"User clicked a Calendar button. Update displayed date and note accordingly."  
selectedDate value: aDate.  
note value: (notes at: aDate ifAbsent: [])
```

As the next step, we will define the *Action* message of the *New date* button. The purpose of this message (we called it `newDate`) is to get a new month and year from the user and display it. To do the display, it will get the subcanvas widget from the builder and send it the `client: message` with an instance of the calendar constructed for this particular month to open the calendar. The definition is as follows:

newDate

```
"Request new date and open calendar on the corresponding month and year with the appropriate note."  
| month year sub |  
selectedDate value:  
  (Date newDay: 1  
    monthNumber:  
      (month := (Dialog request: 'Enter number of month' initialAnswer: ") asNumber)  
      year: (year := (Dialog request: 'Enter year' initialAnswer: ") asNumber)).  
  "Create new Calendar and assign it to subcanvas."  
  newCalendar := Calendar newOnMonth: month  
    year: year  
    from: self  
    message: #selection:..  
  sub := (self builder componentAt: #subcanvas) widget.  
  sub client: newCalendar spec: #windowSpec.  
  self selection: (Date newDay: 1 month: (Date nameOfMonth: month) year: year)
```

Two functions are still unimplemented: saving of notes in a file (button *Save Notes*) and assignment of a new note to the dictionary when the user clicks *Accept* in the note popup menu. We will leave the popup menu until the end of this chapter when we deal with menus, and saving is left as an exercise.

<u>Main lessons learned:</u>

- Whenever possible, design classes so that they can be reused in other applications.
- When a part of an application appears to be a candidate for reuse, design it as a stand-alone application and swap it into a new application as a subcanvas.
- If a plug-in sub-application needs to communicate with its master application, create it with a message whose argument provides a link to the master, and equip the master with communication methods so that the sub-application can 'call back' when necessary.
- When a part of the user interface depends on parameters that vary from one execution to another, it may be appropriate to have the builder construct it programmatically.
- To construct a part of the interface programmatically, use `postBuildWith:` and send `add: aComponent` to `UIBuilder` to add components to a previously painted window background.
- To make a reusable component useful, specify its interface messages as symbols and execute them using `perform:`.
- If the message required by another object is not suitable, use an adapter object.
- Blocks carry along the context in which they are defined. To speed up operation and save memory, this context should be minimal and as many arguments as possible should be defined in the block.

Exercises

1. Modify `newDate` so that the date input field initially displays the date of the first day of the month.
2. Re-implement `Diary` using the other suggested implementation of day buttons and compare the two approaches.
3. Write a code fragment to find all months between 1900 to 1999 that have 31 days and begin on Saturday.
4. None of the `Date` formats available for input fields via the Property Tool looks like the format that we desire. Design a new date format to match our specification. (Hint: See class `InputFieldSpec`.)
5. Write a pluggable adapter with a creation message as in `Translator` from: `symbol1` to: `symbol2` for: `anObject` used as in `Translator` from: `#set` to: `#setIt` for: `anObject`. When the translator with these parameters gets message `set`, it sends message `setIt` to `anObject`. Use this principle to re-implement the `Diary`.

A.2.4 The Notebook widget

In its minimal configuration, the notebook widget is a collection of pages selectable by a tab on the side (Figure A.2.11). Each page is an empty area with a 'book binding' and tabs, and displays a subcanvas determined by the program. Different pages may use different subcanvas definitions or the same canvas definition with different contents. In addition to the required 'major' tabs normally displayed on the side, one can also specify 'minor' tabs along the bottom and change various notebook properties including tab positioning, binding width and location, and colors, either via the Properties Tool or via the builder at run time.

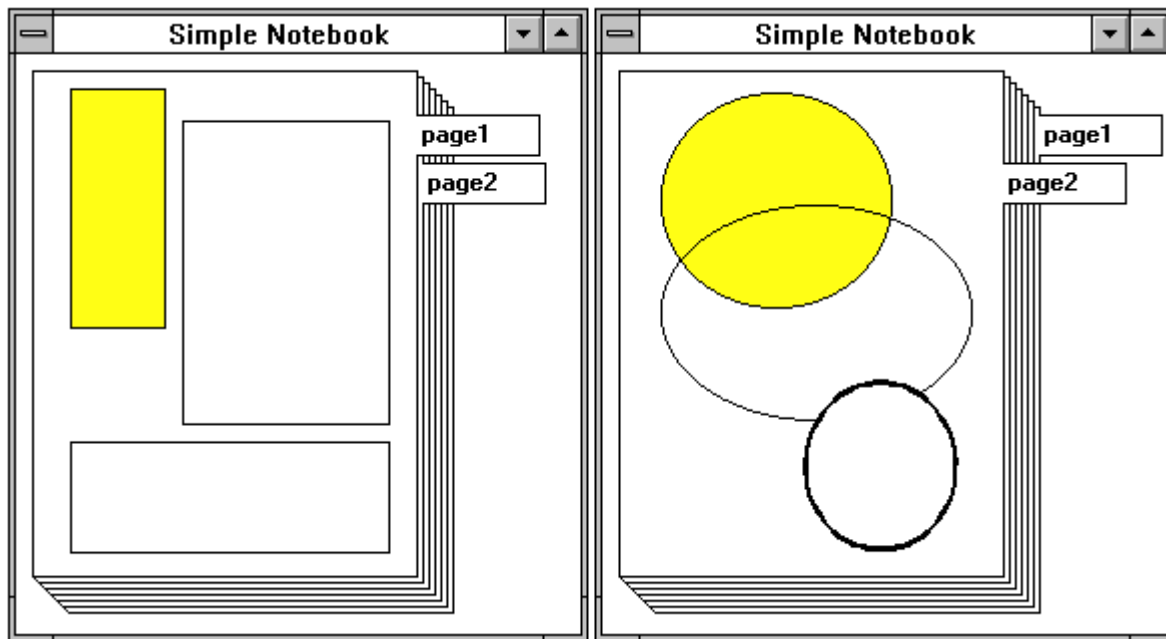


Figure A.2.11. The two pages of the notebook in Example 1.

A notebook with tabs is essentially a single-selection list controlled by tabs, and the tab model is thus a *SelectionInList*. If the notebook uses both major and minor tabs, each set of tabs has its own *SelectionInList* model. To define a notebook widget, paint the notebook on the canvas and assign *Aspect* symbols to major tabs and minor tabs (if required). The *Aspect* is a method that accesses the model of the tabs, a *SelectionInList*. Finally, one must define the swap-in subcanvas or subcanvases. The initialization and change response requirements are as follows:

- initialize
 - assigns a *SelectionInList* with a list of names to the *Aspect* property of the tabs,
 - registers interest in the changes of tab selections.
- postOpenWith: defines the initial tab selection and the corresponding subcanvas. We will see shortly that specifying the subcanvas is not always necessary.
- The change method registered in initialize determines the notebook's response to changed tab selection. If the desired new page uses the same subcanvas as the current page, only the new contents need to be specified. If the new page requires a different subcanvas, the change method must specify it. If the method explicitly assigns the subcanvas, the postOpenWith: method does not have to include subcanvas specification because the change message is automatically sent when the notebook opens.

We will now illustrate the basic procedure on a simple example. A more sophisticated use of notebooks is shown in an application developed later in this chapter.

Example 1: A simple notebook - notebook with major tabs only

Problem: Implement an application consisting of a notebook with the interface shown in Figure A.2.11. The first page contains some rectangles, the second page contains some ellipses, and there are no minor tabs.

Solution: Following our outline, paint a canvas with a notebook widget and specify an *ID* (*#notebook*) for accessing the notebook through the builder to display a new subcanvas, and an *Aspect* for major tabs (*#pages*) to access the *SelectionInList* model. This will make it possible to paint new pages (subcanvases) when tab selection changes.

After painting and installing the canvas (class *SimpleNotebook*), create the two canvases representing the two notebook pages making sure that they will fit into the subcanvas of the notebook, and

install them on the same application model as the main window (we called the window spec methods `#page1` and `#page2`). In our example, the geometric shapes are simply *Region* widgets.

The initialization method assigns a `SelectionInList` to the major tab *Aspect* of the notebook and registers interest in changes in tab selections:

initialize

```
"Define tabs and their labels, register interest in tab changes."  
pages := SelectionInList with: #('page1' 'page2').  
pages selectionIndexHolder onChangeSend: #newPage to: self
```

Method `postBuildWith:` changes tab selection (see below) and sends message `newPage`. The message is, of course, also sent when the user changes tab selection. The message checks the current tab selection and gets the corresponding subcanvas using the appropriate window spec method:

newPage

```
"Tab selection has changed, display the appropriate page."  
| widget |  
"Get notebook widget from the builder."  
widget := (self builder componentAt: #notebook) widget.  
"Display appropriate canvas."  
pages selectionIndex = 1  
    ifTrue: [widget client: self spec: #page1]  
    ifFalse: [widget client: self spec: #page2]
```

Method `postOpenWith:` defines the initial tag selection and this change triggers the `newPage` message which selects the subcanvas displayed when the notebook opens. The definition is as follows:

postOpenWith: aBuilder

```
"Specify initial tab selection."  
pages selectionIndex: 1
```

Example 2: File display - a notebook with major and minor tabs

Problem: Implement an application that allows the user to select a directory, displays the corresponding file names in a notebook, and the contents of the selected file in the adjacent text view (Figure A.2.12).

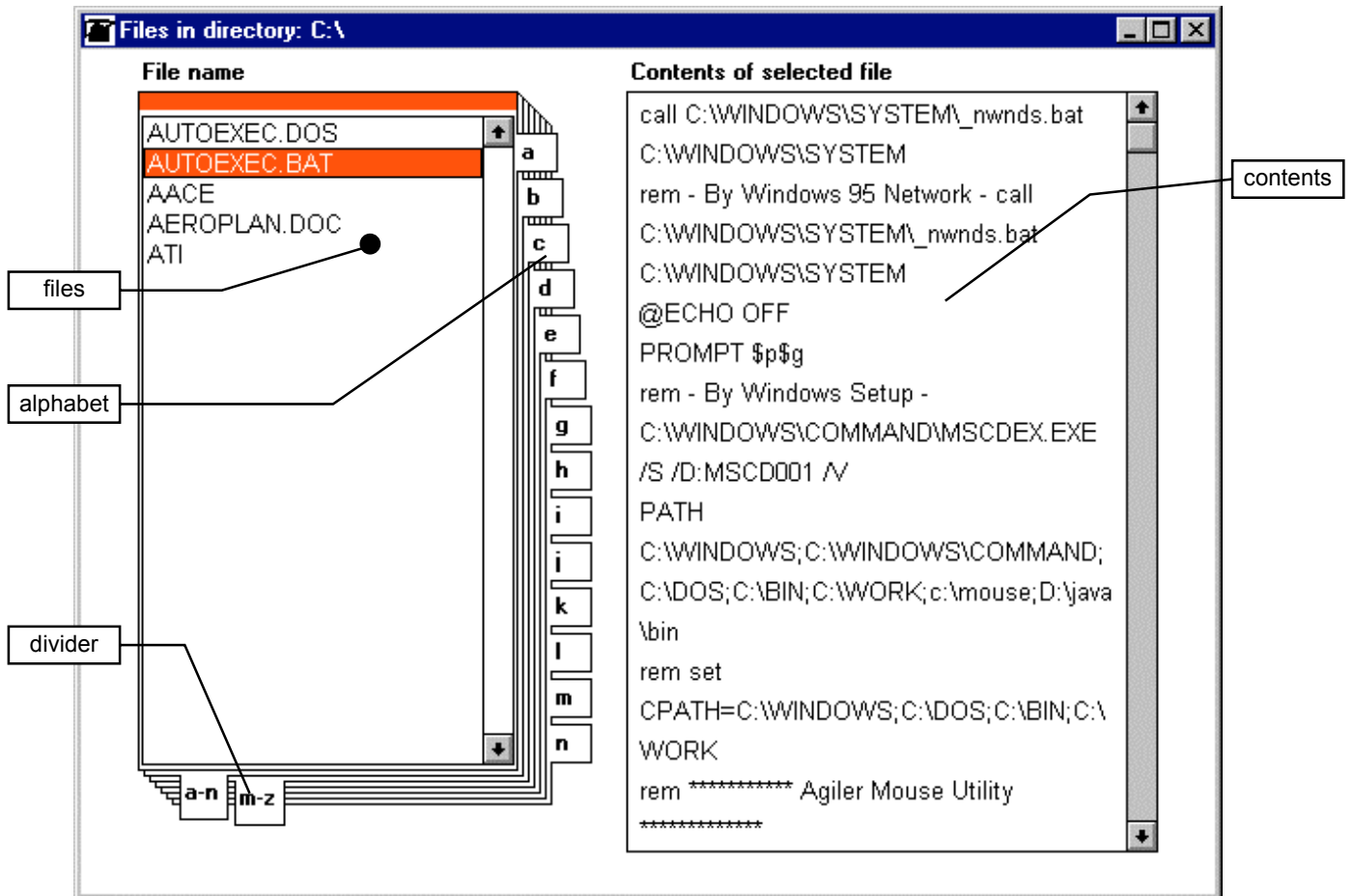


Figure A.2.12. Desired user interface for Example 2 and assignment of widget *Aspect* variables.

Solution: This is a simple tool that can be implemented by a single application model class (FileDisplay) with two window specs – one for the main window (windowSpec) and the other for the subcanvas (subCanvas). The required instance variables include *Aspect* variables for major and minor tabs, the list widget, and the text editor. We will call them *alphabet*, *divider*, *files*, and *contents* respectively as indicated in Figure A.2.12. We will also store the string representing the current directory in variable *directoryString*. Figure A.2.14 shows how the *Aspects* of major and minor tabs are specified in the Properties Tool.

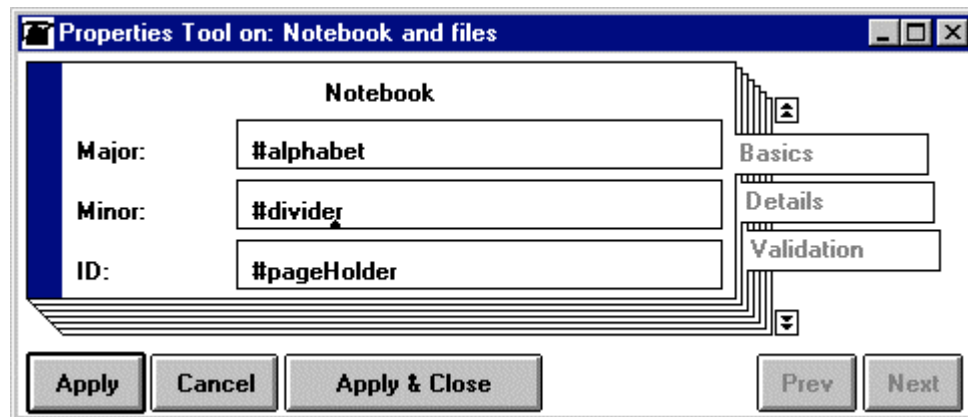


Figure A.2.14. Specifying major and minor tabs for a notebook.

In addition to *Aspect* variables, the program must switch between the two overlapping lists of letters a-to-n and m-to-z corresponding to the two alphabet settings of the major tabs; we will call these lists list1 and list 2. Finally, the file name selected in the list will be used in more than one place and we will assign it another instance variable called filename.

The required methods include initialization, change methods responding to changes in the selection of major and minor tabs and in the filename list, and an *Action* method for the *New directory* button. We must also create the subcanvas containing the list displayed in the notebook. Its specification will be stored in class method subcanvas in the same application model class.

We are now ready for the details and we will start with initialization. Initialization consists of two steps. Method initialize initializes *Aspect* and auxiliary variables, and specifies change messages. Method initialize is as follows:

initialize

```
| offset |  
"Calculate tab labels. They are one-letter strings containing consecutive letters of the alphabet."  
list1 := Array new: 14.  
list2 := Array new: 14.  
offset := $a asInteger - 1.  
1 to: list1 size do: [:index | list1 at: index put: (String with: (index + offset) asCharacter)].  
1 to: list2 size do: [:index | list2 at: index put: (String with: (index + offset + 12) asCharacter)].  
"Define major and minor tabs. Minor tabs are fixed, major tabs depend on the selected minor tab."  
alphabet := SelectionInList with: list1.  
divider := SelectionInList with: #('a-n' 'm-z').  
"Define change messages for notebook tabs."  
alphabet selectionIndexHolder onChangeSend: #changedLetter to: self.  
divider selectionIndexHolder onChangeSend: #changedHalf to: self.  
"The contents of the file list will initially be files in C:\ starting with letter a."  
files := SelectionInList with: ((filename := Filename named: 'C:\') directoryContents  
select: [:fName | (fName at: 1) asUppercase = $A]).  
files selectionIndex: 0. "No file is initially selected."  
"Change of filename list selection triggers a change message."  
files selectionIndexHolder onChangeSend: #newFileSelection to: self.  
"Select the 'a' major tab initially."  
alphabet selectionIndex: 1
```

Note that there is no difference between the principles of treatment of major and minor keys. Assignment of label to the window and assignment of the subcanvas to the notebook page happens in postOpenWith: because it depends on the bindings dictionary:

postOpenWith: aBuilder

```
postOpenWith: aBuilder  
"Assign window label."  
self builder window label: 'Files in directory: c:\'.  
"Assign the file list subcanvas to the notebook page."  
(aBuilder componentAt: #pageHolder) widget client: self spec: #subCanvas
```

Our next definition is the *Action* method of the *New directory* button. It gets the name of a new directory from the user, and checks that it corresponds to an existing directory name. If it does, it first changes the label of the window to show the filename. It then resets major and minor buttons to show letter a, and assigns a new value to the filename list value holder, triggering its redisplay with new file names, all starting with \$A. Via the implicitly invoked change message, this also blanks the text editor view.

newDirectory

```
"Get directory name and switch to it if it is valid."  
| dirString dirName |  
dirString := Dialog request: 'Enter complete filename path as in C: or C:\visual' initialAnswer: 'C:'.
```



```
dirString isEmpty | (dirName := dirString asFilename) isDirectory not
  ifTrue: [^Dialog warn: dirName asString , ' is not a valid directory.']
  ifFalse: [directoryString := dirString].
    self builder window label:
      ('Files in directory: ' , directoryString copyWith: Filename separator).
    divider selectionIndex: 1.
    alphabet selectionIndex: 1
```

Finally the change messages. When the user makes a new selection in the filename list, we must check whether a file is now selected or not and if it is, display it by changing the value holder of the text editor. The display will, of course, work only for ASCII files.

newFileSelection

"A new file has been selected. Display its contents or blank Text Editor for no selection."

```
| text |
files selectionIndex = 0
  ifTrue: [text := ""]
  ifFalse: [text := ((directoryString copyWith: Filename separator), files selection)
    asFilename contentsOfEntireFile].
self contents value: text
```

The change message for major tabs checks whether a tab is selected. If there is no selection, we select tab a because the initial display of the notebook would otherwise cause problems via change messages that depend on tab selection. We then use the value of the major tab to find all files in the current directory starting with the selected letter and assign them to the filename list. This redraws the list and sends a change message (see initialization) that blanks the text editor view.

changedLetter

"User selected new major tab, display new list of files."

```
| path |
alphabet selectionIndex = 0 ifTrue: [alphabet selectionIndex: 1].
path := directoryString copyWith: Filename separator.
files list: (path asFilename directoryContents
  select: [:string | (path , string) asFilename isDirectory not & (
    (string at: 1) asUppercase = (alphabet selection at: 1) asUppercase)])
```

Note the use of Filename separator which makes the path platform independent – different platforms select appropriate separators. Finally the change message for minor keys. When minor key selection changes, major keys must display the appropriate part of the alphabet. The message also selects the first letter of the list as the major tab selection, calculates the corresponding filename list, and assigns it to the list value holder. Through change messages, this modifies the contents of the text editor as well.

changedHalf

"Minor tab selection has changed. Display the alternative list of minor tabs and propagate changes."

```
| letter |
divider selection = 'm-z'
  ifTrue: [alphabet list: list2.
    letter := $m]
  ifFalse: [alphabet list: list1.
    letter := $a].
files list: (filename directoryContents select: [:fileName | (fileName at: 1) asUppercase = letter])
```

Main lessons learned:

- A notebook is a list of pages accessed by tags.
- Up to two sets of tabs can be defined - major and minor. Major tabs are required, minor tabs are optional.
- Each set of tabs uses its own SelectionInList as its model.
- Each notebook page is a subcanvas holder.
- All pages in a notebook may use the same canvas and differ only in the displayed information. Alternatively, different pages may insert different canvases into their subcanvases.
- Flipping pages is achieved by change messages triggered by tag selections.
- Major and minor tabs are handled identically.

Exercises

1. Why didn't we use `postBuildWith:` instead of `postOpenWith:?` Try it.
2. Modify the Diary to use a notebook instead of a text editor for notes.
3. Implement Example 2.
4. Trace change in tab selection.

A.2.5 Dialog windows

A dialog window is the window used by Dialog class messages such as `confirm:` and `warn:`. Its distinguishing feature is that it is *modal* or *pre-emptive*, which means that it does not give up control until it is closed. In other words, you must finish all interaction and close the window before you can interact with another window. This behavior is implemented in class `SimpleDialog`, a subclass of `ApplicationModel`. In this section, we will explain how to create your own dialog window. Although modal windows provide an interesting and sometimes necessary behavior, user interface experts discourage their indiscriminate use because they restrict user's freedom.

There are several ways to make a window pre-emptive. One way is to *install* a canvas as a *Dialog* instead of an *Application*, making its model a subclass of `SimpleDialog` instead of `ApplicationModel`. Another possibility is to *install* the canvas as *Application* (its model is thus a subclass of `ApplicationModel` as usual) but *open* it as a dialog. You can also construct the dialog programmatically from scratch using `SimpleDialog` rather than painting the canvas with the UI Painter; this is how dialogs in class `Dialog` are implemented. Finally, you can change the same window from modal to non-modal and vice versa programmatically. Each of these methods has its advantages and disadvantages and we will now give a simple example to illustrate the first two approaches which are most common.

Example: A password dialog window

Problem: A library application requires a password. Create a dialog window that looks like the `request:InitialAnswer:` window but shows asterisks as the user types the text in (Figure A.2.14). The dialog returns the text entered by the user when the user clicks *Accept*, or an empty string when the user clicks *Cancel*.



Figure A.2.14. Desired look of password dialog window.

Solution 1: Creating a dialog with the UI Painter and installing it as a Dialog

In this implementation, we will create the password dialog as a separate class called `PasswordDialog`. Creating its user interface is very similar to creating the usual *Application* canvas: Paint the window choosing *password* style and aspect text for the input field, *Install* it on the model class as usual, but specify *Dialog* as the window type (Figure A.2.15). As the figure shows, this makes the application model a subclass of `SimpleDialog` (which is a subclass of `ApplicationModel`.)

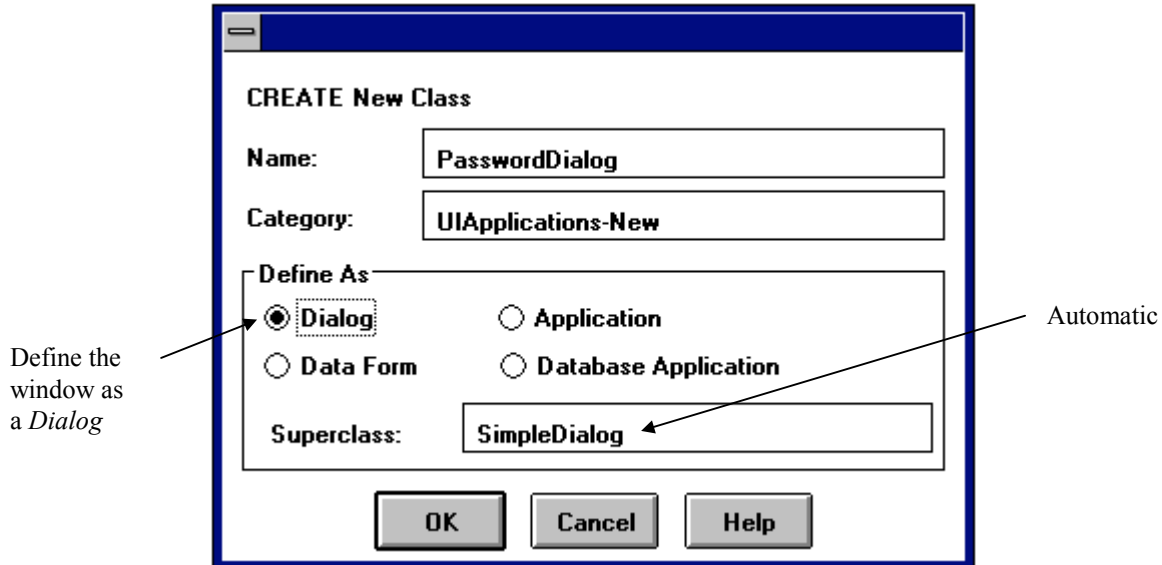


Figure A.2.15. A dialog canvas can be created by defining it as a *Dialog*.

The rest is simple except that we must deal with the fact that when the user closes a modal window, the window can return only `true` (built-in and unchangeable behavior of method `accept`) or `false` (built-in and unchangeable behavior of method `cancel`). As a consequence,

```
password := PasswordDialog password
```

can only return `true` or `false`, no matter what the user entered into the dialog window. We thus need a way for the dialog window to communicate its acquired data back to `Library`. One way to obtain this information is to create a `PasswordDialog` with a message that tells it who is the sender and what is the message to be used to return the appropriate information. (We used the same approach with subcanvases.) As an example, the opening message could be

```
openFrom: anObject onClosing: aMessageSymbol
```

and the method in class `Library` that obtains it will be, for example,

getPassword

```
PasswordDialog openFrom: self onClosing: #password:
```

Assuming this definition, the behavior of the dialog window is as follows: When the user closes the password dialog window, the dialog window closing method will send the `password:` message to the `Library`

object with the string entered by the user as its argument (Figure A.2.16). Its argument will be the password if the user clicked *Accept*, and an empty string if the user clicked *Cancel*.

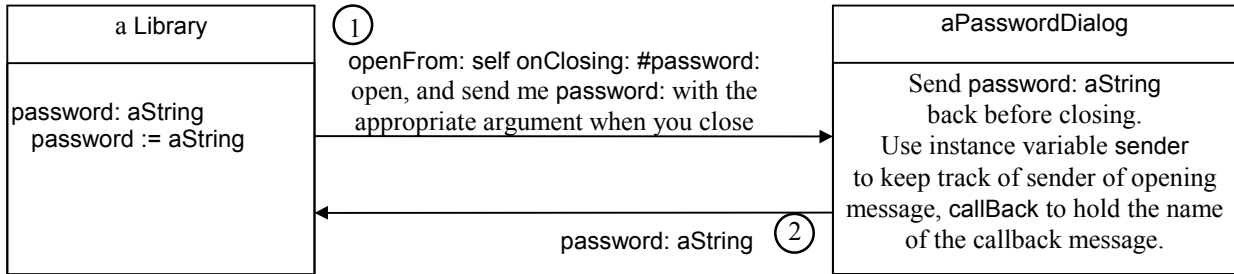


Figure A.2.16. Returning results of a dialog.

The definition of the PasswordDialog opening message is as follows:

openFrom: anObject onClosing: aSymbol

"Open password window , get user input, return result to anObject using message aSymbol."
^(self new sender: anObject; callback: aSymbol) open

where sender: and callback: are accessing message accessing instance variables sender and callback:.

To implement this solution, we paint the user interface of our password window, define the *Action* properties of *Accept* (acceptAndClose) and *Cancel* (cancelAndClose) buttons, and install the canvas on class PasswordDialog as a *Dialog*. Note that it is not enough to redefine accept and cancel because instances of SimpleDialog and its subclasses always execute the pre-defined accept and cancel methods and ignore any other definitions.

Method acceptAndClose will send callback to the sender with the string entered by the user as the argument. Method cancelAndClose will send callback with an empty string. Both methods conclude by sending closeRequest to close the window:

acceptAndClose

"Return password to sender and close window."
sender perform: callback with: text value.
self closeRequest

cancelAndClose

"Return empty string to sender and close window."
sender perform: callback with: ".
self closeRequest

where sender is an instance variable of class PasswordDialog initialized to the argument of openFrom: aSender onClosing: aSymbol when the window opens. Note that the callback message has an argument.

To test our solution, define Library with method getPassword as explained above, one instance variable called password, and accessing methods password and password:. To test the dialog window, execute the following expression with *inspect*:

```
Library new getPassword; password
```

The password window opens, the inspector opens on PasswordTest, and the returned value of password is as expected.

Solution 2: Installing a dialog as an Application and opening it as a dialog

In this approach, we will install the canvas on `Library` which is an `ApplicationModel`. The new interface will be stored in a window spec method called `passwordWindow` but since the superclass is not `SimpleDialog`, we will have to open it with `openDialogInterface: #passwordWindow` rather than `open`. The new definition of `getPassword` will thus be

```
getPassword  
self openDialogInterface: # passwordWindow
```

Closing the window will again return `true` or `false` and we thus must again define `acceptAndClose` and `cancelAndClose` methods for the *Accept* and *Cancel* buttons. This time, we will not send `closeRequest` to close the window because we don't want to close the application but only the password window. To do this, the closing methods will get the current window (the password window), and ask its controller to close the window². In addition, the closing methods will assign an appropriate value to `password`. The definitions are

```
acceptAndClose  
password := text value.  
Window currentWindow controller close
```

and

```
cancelAndClose  
password := text value.  
Window currentWindow controller close
```

Finally, note that although we implemented the dialog as a part of `Library`, we could have implemented it as a standalone class just as in Solution 1.

Solution 3: Creating the dialog programmatically via SimpleDialog

Class `SimpleDialog` contains numerous utility messages for building custom dialog windows programmatically and class `Dialog` uses these messages to build its utility dialogs. The advantage of this approach is that it provides full control over the behavior of the window, including the returned object. The disadvantage is that it requires more complex programming. We will not demonstrate the procedure but if you wish to see the details, browse `request:initialAnswer:onCancel:for:` in class `SimpleDialog`. This method is the basis of the implementation of the familiar `request:initialAnswer:` method in class `Dialog`.

Summary of the three approaches

The most common way of creating modal dialogs is to paint the dialog window with the UI painter and install it as a *Dialog* or as an *Application*. In both cases, closing the window with `accept` or `cancel` returns `true` or `false` and if we want to pass another result to the sender of the dialog opening message, we must take extra measures: If the canvas is installed as a *Dialog*, its class needs to know who is sending the opening message and how to pass the result back. The advantage of this approach is that the dialog is reusable in many situations. If the canvas is installed as an *Application*, the sender usually is the application model itself and the methods that `accept` or `cancel` the result of the dialog can access any of the variables of the application directly. This approach is thus somewhat simpler but the dialog is not reusable because it is a part of the application model.

Main lessons learned:

- A dialog is a modal (pre-emptive) window that does not relinquish control until it is closed.
- The use of modal windows is discouraged because they restrict user's freedom.

² For more about controllers, see Chapter 12.

- There are three ways to create dialogs in VisualWorks: By painting the canvas and installing it on a subclass of *SimpleDialog* or a subclass of *ApplicationModel*, or by constructing the window programmatically as a subclass of *SimpleDialog*. The first two approaches are easier and more common, the third approach provides more control.
- When installing a dialog as a *Dialog*, define a creation message that identifies the sender and the message to be used to return the result. When the dialog is installed as an *Application* (but opened as a *Dialog*), dialog closing messages can access the application model directly.
- Installing a dialog as a *Dialog* makes the dialog reusable. Installing a dialog as an *Application* and opening it as a *Dialog* is simpler but it ties the dialog interface to the application.
- The nature of a window (application or dialog) can be changed at run time.

Exercises

1. You may have noticed that if you specify text as the argument for Dialog prompts such as `warn:`, the emphasis does not have any effect. Find why and modify the code to remove this restriction.
2. One slightly annoying feature of built-in dialogs from class *Dialog* is that the dialog windows appear at the latest location of the cursor. Define a new class called *PositionableDialog* with methods replicating the behavior of `warn:`, `confirm:`, and `request:initialAnswer:` but adding position control as in `confirmAtPositionFromUser:`. (Hint: The opening message `openAt: aPoint` allows the programmer to specify where the dialog window should open.)
3. Implement similar modifications as in Exercise 2 but open the dialog window in the center of the screen. (Hint: Class *Screen* provides access to the properties of your computer screen.)

A.2.6 Menus in general and popup menus in particular

A menu is a collection of labels and clicking a label returns a value, possibly calculated by an expression or a block associated with the label. Functionally, a menu is thus a collection of label-value pairs.

The UI Painter provides three types of menus: popup menus, menu bars, and menu buttons. Popup menus that you can build are the familiar menus invoked by the `<operate>` button, an example of a menu bar is the list of commands at the top of the Launcher, and a menu button is a drop-down menu that looks like an input field when inactive but drops down a menu when activated (Figure A.2.17). The purpose of menus is similar to the purpose of single-selection lists - selecting one of several items - and their advantage is that they occupy little or no window space. The advantage of selection lists is that they make the choices visible at all times.

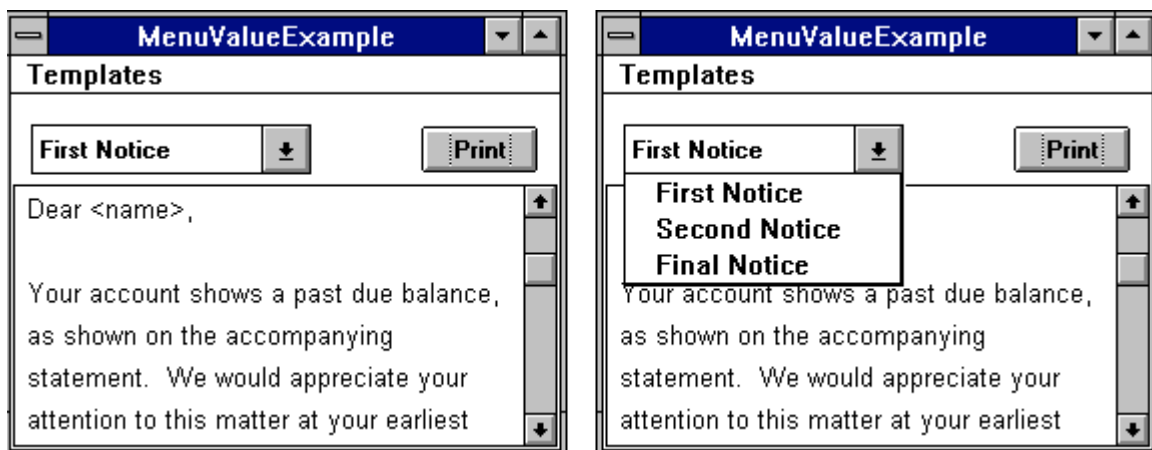


Figure A.2.17. Passive (left) and activated (right) menu button. The passive view may display the current choice in the menu. The menu bar at the top of the window has one label.

The distinguishing characteristics of popup menus, menu bars, and menu buttons are summarized in the following table:

type of menu	behavior	type of GUI component	placement
menu bar	drop-down	window property	top of window, no program control
menu button	drop-down	stand alone widget	any place in window
popup menu	popup	widget property - <operate> menu	pops up at cursor location

All types of menus depend on class `Menu` which holds information about instances of `MenuItem`. The protocol of `Menu` deals with the menu as a whole and provides methods for adding and removing menu items, hiding them, changing the menu's background color, and accessing individual menu items by their label. Once a menu item is retrieved, the `MenuItem` protocol can be applied to it, for example to define the emphasis of the text of the label. A `MenuItem` may also be a collection of menu items – a submenu – and this recursive structure allows a menu to contain submenus nested to any depth.

Creating menus

A menu can be created in the following ways:

- Using `Menu` and `MenuItem` directly. This approach provides most flexibility but it is also most laborious because every detail must be programmed. Use this style only if necessary, for example, if you must use special fonts for labels, change the background color, etc.
- Using an instance of `MenuBuilder`. This class has a number of convenience methods that are useful to build menus with limited programming. An instance of `MenuBuilder` is exactly what its name suggests – a tool that builds a `Menu` object. Once built, this object can be accessed in all ways available through `Menu` and `MenuItem` protocols.
- Using the Menu Editor. A Menu Editor is a painting tool that eliminates the need to program the contents of the menu explicitly. Its purpose is similar to that of the UI painter – to minimize programming. Being a substitute for programming, it is also less flexible than other methods of menu creation and cannot be used, for example, to create menus that change at run time. Use the Menu Editor when the menu remains unchanged at run time and uses default settings.

The following is a comparison of the three approaches:

Name	Creation method	Use	Flexibility	Accessing	Programming effort
<code>MenuBuilder</code>	program	dynamic menus	medium	instance method	medium
<code>Menu</code> , <code>MenuItem</code>	program	dynamic menus	full	instance method	large
Menu Editor	paint	static menus	limited	class method	minimal

In this and the following sections, we will now give several examples of creation of simple popup menus and menu bars, leaving menu buttons as exercises.

Popup menus

Popup menus are attached to the <operate> button and can be specified for certain widgets including input fields, selection lists, and text editors. They should be used only for commands that are directly related to the widget. Although this sounds natural, many designers don't follow this rule and put, for example, a command to close the *window* into the popup menu of a list widget contained in the window. This is illogical and confuses the user.

Each widget that allows a popup menu has a default menu. The default popup menu of selection lists is no menu whereas the default popup menu of input fields and text editors is the familiar text editing menu with *copy*, *cut*, *paste*, and other commands. To create a popup menu different from the default, enter the name of the method that returns the menu into the widget's *Menu* property (Figure A.2.18).

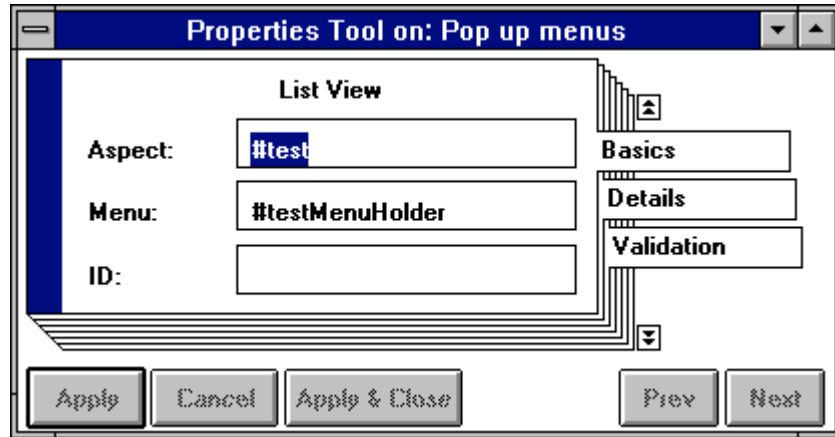


Figure A.2.18. To create a non-default menu, enter the name of the method that returns the menu in the *Menu* property.

We will now give two examples that show how to create a popup menu using the *Menu Editor* tool and programmatically.

Example 1: Creating a menu with the Menu Editor tool

The Menu Editor is suitable for creating popup menus that don't change while the application is running. The specification of a menu created with the Menu Editor is stored in a *class* method that describes the menu, somewhat like the window specification method.

Problem: Create a popup menu for the selection list in Figure A.2.19. Clicking *add* will open a dialog allowing the user to enter a new string which will then be added to the selection list and displayed. Clicking *delete* will delete the label currently selected in the selection list, if any.

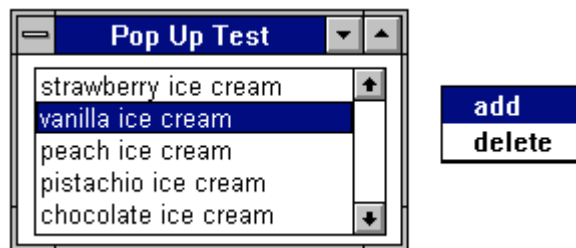


Figure A.2.19. Window containing a selection list with a popup menu.

Solution: Paint the canvas, install it on an application class (ListWithPopup1), and *Define* the properties. (We called the *Aspect* of the list list, and its *Menu* property listHolderMenu.) Since the popup menu for the list does not change during execution, we will create it with the Menu Editor tool. Open it from the *Tools* command in the *Canvas* tool or from the <operate> menu of the canvas, and enter the label-command pairs (Figure A.2.20). The symbol entered for *Value* is the name of an instance method in the application model that will be executed when the user clicks the label highlighted on the left.

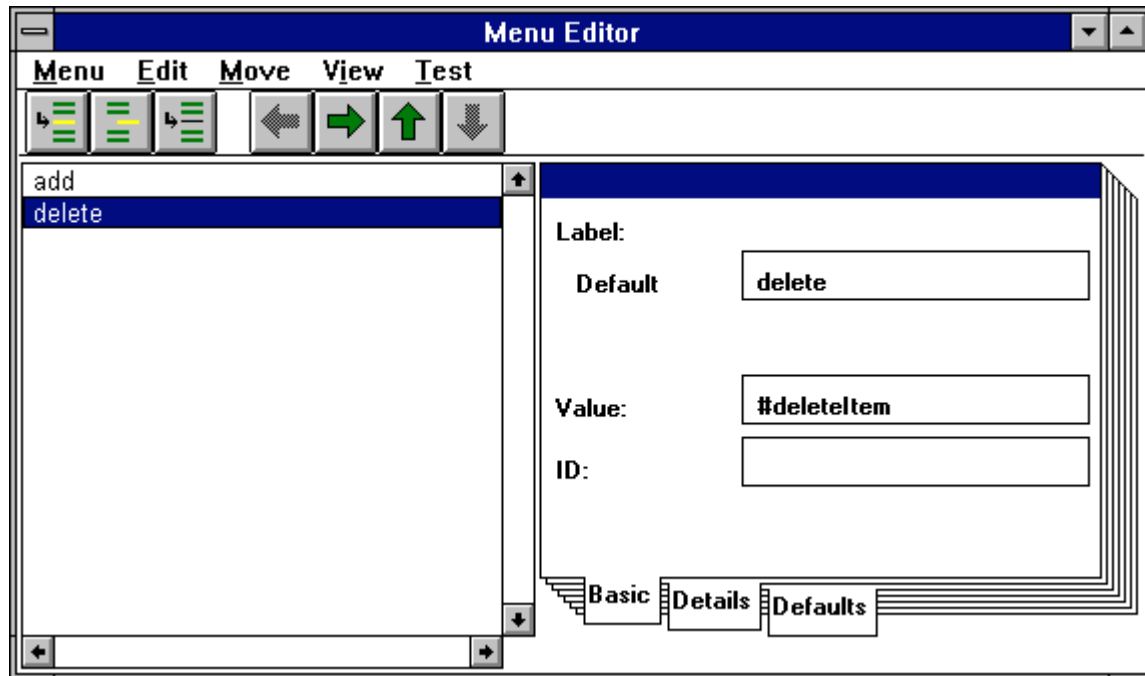


Figure A.2.20. Using Menu Editor to create a menu.

After entering the two label-command pairs (*add* – *addItem*, *delete* – *deleteItem*), click *Install* under the *Menu* command of the Menu Editor. This will open the dialog in Figure A.2.21. For *class*, enter the name of the menu-accessing *method* previously specified as the *Menu* Property of the selection list, and click *OK*. This creates the resource method describing the menu, and stores it in the *resources* class protocol. By clicking *Test* in the Menu Editor, you can now examine what the menu looks like.

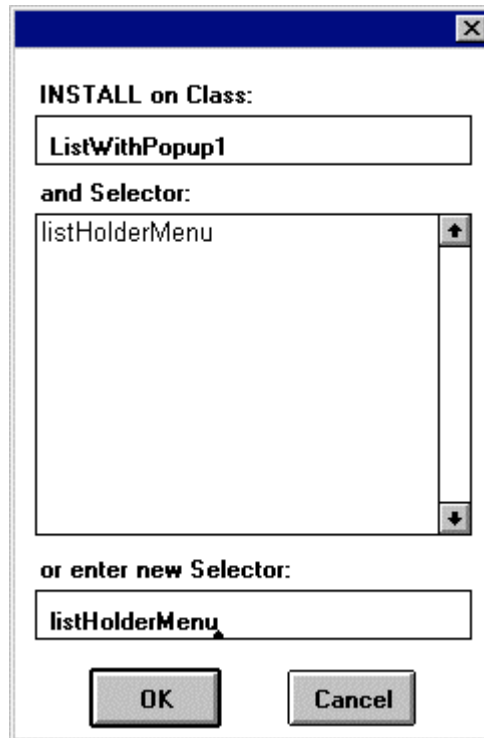


Figure A.2.21. Dialog for installing a menu.

In our case, the automatically generated resource definition method is

testMenuHolder

```
"MenuEditor new openOnClass: self andSelector: #testMenuHolder"
<resource: #menu>
^#(#Menu #(
    #(#MenuItem
        #rawLabel: 'add'
        #value: #addItem )
    #(#MenuItem
        #rawLabel: 'delete'
        #value: #deleteItem ) ) #(2 ) nil ) decodeAsLiteralArray
```

and executing the comment line opens a Menu Editor on the menu.

You can now open your application and test the popup menu. The *add* and *delete* commands will appear when you press the <operate> mouse button but they will not do anything, of course, because we have not defined the corresponding messages yet. We will define these methods now.

The method implementing the *add* command gets a new label from the user, adds it to the list of labels, and assigns it to the list widget via list:

addItem

```
"Ask user for new label and add it to the list widget."
list list: (list list add: (Dialog request: 'Enter new label' initialAnswer: ""))
```

where the misuse of identifier list is almost not funny any more. To test the method, we opened the dialog window, selected *add*, but when we entered 'label 1', the result was as in Figure A.2.22. This is not what we desired - and a typical problem.

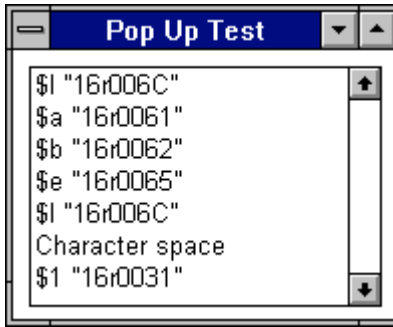


Figure A.2.22. Result obtained with the first version of `addItem`.

The error clearly occurs when the method adds the new string to the list. Everything looks OK so we inserted a `halt` message into `addList` and traced its execution. We found that when we `add:` the new label, the list part of the selection list ceases to be a `List` and becomes a `ByteString`. We now realize that the `add:` message does not return the modified collection (the list) but the argument - in this case the result of the Dialog expression. To correct this, we must send `yourself` after sending `add:`.

addItem

```
list list: (list list add: (Dialog request: 'Enter new label' initialAnswer: '')); yourself)
```

This version works correctly.

Finally the definition of the method executing the *delete* command. The method first checks whether the user made a selection. If not, the method does not do anything. If yes, the method gets the current list, removes the current selection from it, and assigns the new list via `list:`.

deleteItem

```
| selection |  
selection := list selection.  
selection isNil ifTrue: [^self].  
list list: ((list list) remove: selection; yourself)
```

Example 2: Creating and changing a menu dynamically

Problem: The user interface in Example 1 is less than perfect: When no item is selected, the user cannot delete an item and the menu should not offer the *delete* command. For proper operation, we need two different popup menus - one with *delete* and one without it - and display the menu appropriate for the situation. Such a dynamic menu cannot be created with the Menu Editor.

Solution: Although the mechanics of the required code is trivial, a minimal understanding of the basics of menu operation is useful and we will thus start with some background information.

As you already know, active widgets consist of the visual part displayed on the screen (the *view*), and the object that tracks user input and responds to it (the *controller*). If a widget has a popup menu, the widget's controller must know what this menu is because the menu pops up in response to mouse button activation and its operation also depends on user input. To be able to do this, the widget's controller is an instance of `ControllerWithMenu` which holds the widget's `Menu` object in an instance variable and if we want to change the menu at run time, we must thus change the contents of this instance variable.

The easiest way to change the `Menu` object is to put it into a value holder and control its contents via `value:`. In our case, the menu changes whenever the user selects or deselects an item in the list, and to implement this behavior, we will register our interest in any change of list selection via `onChangeSend:to:`. When a change of selection occurs, the change message will send `value:` with the appropriate menu to the menu holder. These arrangements will be made in the initialization method which will assign the initial menu to the menu holder and register interest in selection changes:

initialize

```
"Assign initial popup menu to the list and register interest in selection changes."  
listHolderMenu := self menuWithoutDelete asValue.  
list := SelectionInList new.  
list selectionIndexHolder onChangeSend: #changedMenu to: self
```

The `changedMenu` method will check whether the list has a selection and supply the menu without *delete* if the list has no selection, or the full menu if there is a selection:

changedMenu

```
"Context may have changed - assign the appropriate popup menu."  
self listHolderMenu value: (list selection isNil  
    ifTrue: [self menuWithoutDelete]  
    ifFalse: [self menuWithDelete])
```

where `listHolderMenu` is the *Menu* method that we specified in the Properties of the list widget:

listHolderMenu

```
^listHolderMenu
```

This method is also executed by the builder when it builds the list and its controller. The method that constructs the full menu uses the `MenuBuilder` and its definition is as follows:

menuWithDelete

```
"Calculate menu with add and delete commands."  
| menuBuilder |  
menuBuilder := MenuBuilder new.  
menuBuilder add: 'add' -> #addItem;          "Label-value pair."  
             add: 'delete' -> #deleteItem.  
^menuBuilder menu
```

This method demonstrates the standard usage of a `MenuBuilder`:

1. Create a new `MenuBuilder` object to build the menu.
2. Provide menu information to the `MenuBuilder` via `add:` and possibly other building messages.
3. Ask the builder to create and return a `Menu` object via `menu`. The `MenuBuilder` uses the information gathered in Step 2 to build a `Menu` object containing `MenuItem` objects and other information. Don't forget to return the menu.

Method `menuWithDelete` is very much the same and we leave it as an exercise. All other methods remain as in Example 1 but the class method `listHolderMenu` should now be deleted because it is not needed any more.

Finally, any menu – whether created programmatically or with Menu Editor – is held in a menu holder in the *Aspect* variable specified as the *Menu* property of the list. Consequently, menus created with Menu Editor can be also be changed programmatically. The disadvantage is that once the default menu is replaced, it is not accessible as easily and dynamically changing menus are thus normally created as described in this example.

Overriding the Text Editor and Input Field default menu

The Text Editor and the Input Field have a pre-assigned popup menu with defined behaviors. As an example, *accept* simply assigns the text to the widget's *Aspect* variable and cannot be easily redefined. Sometimes, we might want to change these behaviors. In our Diary, for example, we would like *accept* to add the text in the Text Editor to notes. To do this, we must modify the *accept* command in the built-in menu and this requires a new menu which can be created as follows:

1. Specify the name of the method that returns the new menu as the *Menu* property of the Text Editor. We will call this menu calculating method *newMenu*.
2. Write the menu creating method (*newMenu* in our case).
3. Write the method executing the menu command (we will call the method that executes *accept saveNote*).

Method *newMenu* builds a new menu as follows:

newMenu

"Redefine *accept* by building a new menu executing non-default menu methods."

```
| mb |
mb := MenuBuilder new.
mb    "First add predefined command combinations with default behaviors."
    addFindReplaceUndo;
    line;
    addCopyCutPaste;
    line;
    "Now add customized commands."
    add: 'accept' -> #addNote;      "This is new."
    add: 'cancel' -> #cancel.      "This is standard."
^mb menu
```

and message *addNote* executed by *accept* simply adds a new association consisting of the selected date and the new note to the notes dictionary:

addNote

"Save new note in notes dictionary."

notes at: selectedDate value put: note value.

Main lessons learned:

- Menus have the advantage that they don't consume window space.
- UI painter menus include popup menus, menu bars, and menu buttons.
- Popup menus are activated by the <operate> button, menu bars are drop-down menus attached to the window, and menu buttons resemble input fields but provide drop-down menus.
- The basis of all menus is *Menu* containing *MenuItem* objects and other information.
- Menus are usually created with the Menu Editor tool or programmatically via a *MenuBuilder*.
- The Menu Editor creates a menu stored as a resource specification and limits programming to writing methods implementing menu commands. Such a menu is not intended to be changed at run time.
- Dynamically changing menus can be programmed using *MenuBuilder*.
- *MenuBuilder* constructs a *Menu* object via convenience messages.
- Controllers of widgets with a popup menu have an instance variable containing a value holder with the menu. This variable is specified as the *Menu* property of the widget.
- When adding a new command to a menu, a common mistake is to forget that *add:* returns its argument.
- Text Editor and Input Field have popup menus with default behaviors. To change the behavior, build a new menu using *MenuBuilder*.

Exercises

1. Class *Menu* has a rich set of controls over the form of the label and the way in which menus are constructed. Write a description of *Menu* and *MenuItem* with detailed information about three selected protocols and five selected methods.
2. The Menu Editor provides a number of interesting options such as adding an image to a menu item, indenting, allowing ID access to a menu item, and so on. Write a summary of its functions. (Hint: See on-line help.)

3. Use the Menu Editor to color the background of the *delete* command in our example red and the *add* command green.
4. Add command *edit* to the popup menu in Example 2. The menu should show it only when a label is selected and it should allow the user to change the selected list item via a dialog.
5. Instead of removing the *delete* command when there is no selection, disable it. (Hint: See the example class protocol in class Menu.)
6. Modify the Text Editor menu in Diary to display *accept* in boldface, and *cancel* in red.
7. Track how a menu holder is accessed during the opening and operation of an application.
8. Find how the builder determines whether to use a class method or an instance method to construct a menu.
9. Track and describe the complete sequence of events that occur when a popup menu is activated by pressing the <operate> mouse button.
10. Find where the <window> popup menu is defined and modify it to display the *close* command in red.

A.2.7 Menu Bars

A menu bar is a set of labels at the top of a window and their associated drop-down menus (Figure A.2.23) called submenus. To assign a menu bar to a window, enter the name of the method returning the menu as the *window's Menu* property. In other respects, menu bars are just like popup menus.

Example: Window with a menu bar

Problem: Implement a window with a text editor and a menu bar as in Figure A.2.23. The *File* command's submenu will contain two commands called *Print* and *Exit*. The *Print* command will print the contents of the text view in the Transcript, and the *Exit* command will close the window. The *Configure* command's submenu will also contain two commands - *Hide Text* and *Show Text*. When the window opens, the text view will be visible, *Hide Text* enabled, and *Show Text* disabled. Clicking *Hide Text* will make the text view invisible, enable *Show Text*, and disable *Hide Text*. Clicking *Show Text* will have the opposite effect.

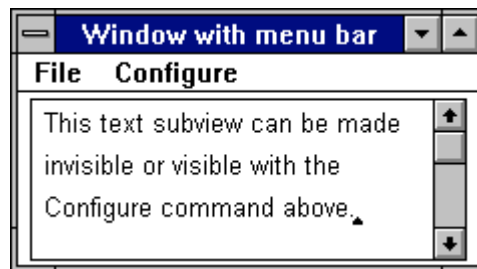


Figure A.2.23. Window with menu bar and text subview.

Solution: Paint the window, enable its *Menu* property, and provide the name of the menu-returning message (*menuBar*) as its *Aspect*. Paint the text view and define its properties (*Aspect* text) including an ID (*#text*) so that we can control its state.

The next step is to define the methods. We will start with the window's *Menu* method *menuBar* which creates the menu bar with its submenus. A static menu can be built the Menu Editor, remembering that the menus are submenus and using the left- and right-pointing arrows in the Menu Editor to achieve the layering. In this example, however, we will create the menu using the *MenuBuilder* to illustrate how to create submenus programmatically.

menuBar

```
"Create instance of Menu for the menu bar."
| menuBuilder |
"Create instance of Menu for the menu bar."
| menuBuilder |
"Create a MenuBuilder."
menuBuilder := MenuBuilder new.
"Use it to construct submenus."
menuBuilder beginSubMenuItemLabeled: 'File';
    add: 'Print' -> #print;
    add: 'Close' -> #closeRequest;
    endSubMenuItem.
menuBuilder beginSubMenuItemLabeled: 'Configure';
    add: 'Show text' -> #showText;
    add: 'Hide text' -> #hideText;
    endSubMenuItem.
"Ask MenuBuilder to construct and return the menu."
^menuBuilder menu
```

The next step is to write the two methods that enable and disable the *show* and *hide* commands in the drop-down menu. In essence, this consists of extracting the proper components of the menu and changing them. The details are as follows: Ask the builder for the menu bar using the *Menu* property method of the window, extract the submenu MenuItem of the *Configure* label, and access its menu items. As an example, the *enableHideText* method which enables the *Hide text* command and disables *Show text* is

enableHideText

```
"Enable 'Hide text' command and disable 'Show text' command."
| menu submenu |
"Get menu bar menu."
menu := self builder menuAt: #menuBar.
"Get the appropriate submenu."
submenu := menu valueForMenuItem: (menu menuItemLabeled: 'Configure').
"Modify the appropriate menu items."
(submenu menuItemLabeled: 'Show text') disable.
(submenu menuItemLabeled: 'Hide text') enable
```

The *enableShowText* method which enables the *Show text* command is similar and we leave it as an exercise.

In the process of opening the application, we must enable *Hide* and disable *Show* after the window has been built but before it opens using the *postBuildWith:* method. We do this by sending the *enableHideText* message as follows:

postBuildWith: aBuilder

```
self enableHideText
```

If you now open the window, it will have the correct menu bar and its commands will have the correct submenus. However, the *Configure* commands will not work because we have not defined the methods that show and hide the text subview widget. These definitions simply get the text editor from the builder, send it *beInvisible* or *beVisible*, and toggle the *Configure* commands. As an example, the *hideText* method is as follows:

hideText

```
(self builder componentAt: #text) beInvisible.
self enableShowText
```

We leave the remaining methods to you as an exercise.

Main lessons learned:

- Menu bars are created and used in much the same way as popup menus but a menu bar is a property of the canvas, and the drop down menus are classified as its submenus.

Exercises

1. Construct the menu bar in our example using the Menu Editor.
2. Draw a diagram showing the structure of the menu bar object being accessed by `enableHideText`. (Hint: Start your search from the builder object.)
3. Modify the *Print* command to open another submenu with two options – *Transcript* (to print to the Transcript) and *Printer* (to print to the printer).

Conclusion

In the first part of this chapter, we introduced datasets. Datasets look like tables but whereas tables can display heterogeneous objects, dataset display rows of objects of the same kind. Unlike tables which are read-only widgets, datasets cells may be editable input fields and other types of widgets.

We then introduced a powerful widget called a subcanvas. A subcanvas is essentially a place holder for a canvas that can be used for swapping of assemblies of UI widgets at run time and for reuse of complete subapplications. Subcanvases are also essential for notebook widgets. While talking about subcanvases, we showed how to use `UIBuilder` to build user interfaces at run time.

Notebooks are widgets that allow access to pages displaying subcanvases. Individual pages may use the same subcanvas and change only its contents, or they may display completely different subcanvases. Access to pages is controlled by major tabs (required) and minor tabs (optional). Tabs are essentially selection lists and both major and minor tabs are handled in the same way.

The next section introduced dialog windows. Dialog windows are modal windows which means that they keep input focus until closed, disabling access to other windows. Because of this imposition on the user, dialog windows are generally discouraged but their use is sometimes necessary. VisualWorks dialog windows are based on class `SimpleDialog` and can be created with the UI Painter. They can be implemented either by specifying their type as *Dialog* instead of *Application* during installation, or by specifying their type as *Application* and opening them with a dialog opening message.

When a VisualWorks dialog window closes, it returns either `true` or `false` which is usually unsatisfactory because dialog windows are typically used as forms for entering data. To get around this, the dialog window must either maintain a link to the application class that opens it (if this class is different from the class of the dialog window), or it must be defined in the application class itself and access its instance variables.

In the last part of this chapter, we covered two varieties of menus: popup menus and menu buttons. A third type of menu is a widget called a menu button. The purpose of menus is similar to that of single-selection lists and radio buttons but they occupy less window space. Their disadvantage is that they don't show the choices at all times. All kinds of menus are created in essentially the same way but whereas popup menus and menu bars are associated with windows or widgets, menu buttons are stand alone widgets. When a menu remains the same during the life time of the application, create it with the Menu Editor tool; when it depends on context, construct it at run time.

Text Editor and Input Field are equipped with default popup menus. To change them or to modify their built-in behavior, replace the default menu with one constructed by Menu Editor or `MenuBuilder`.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Menu, **MenuItem**, SimpleDialog.

Widgets introduced in this chapter

Dataset - display resembles tables but internal implementation is based on lists. All rows are instances of the same domain object and all cells in a column thus share the same type which may be selected from input field, read only, combo box or check box.

Menu Bar - drop down menu attached to a window. Its components are called submenus and may be nested.

Menu Button - drop down menu widget.

Multiple-Selection List - list allowing any number of selections at a time.

Popup Menu – menu activated by a mouse button. Certain widgets may have associated popup menus which are invoked by sending a menu-constructing method specified as the widget's property.

Single-Selection List - list allowing only one selection at a time.

Subcanvas – a canvas holder widget. Can be used to include a subapplication as a part of window or switch parts of a window at run time.

Terms introduced in this chapter

adaptor - object inserted between two communicating objects, typically for the purpose of translating messages from the form used by the sender to the form used by the receiver

call back - communication from an object or a program back to the object or program that created it

controller - the part of a widget that interact with the user

view - the visual part of a widget

Appendix 3 – Chess board – a custom user interface

Overview

This appendix shows the details of the design and implementation of an application with a custom UI with its specialized view and controller. It introduces several new techniques for creating and controlling cursors, view updating, and other useful tasks.

A.3.1 Chess – a specification

In this and the following sections, we will develop a somewhat limited version of a chess front end, as an example of an application using a non-trivial custom widget. The specification of the problem is the subject of this section.

Specification

This program will be a pilot version of a program allowing two players to play chess. Eventually, the players may be two humans or one human and one computer program but the pilot version is limited to the design and implementation of the user interface for two human players.

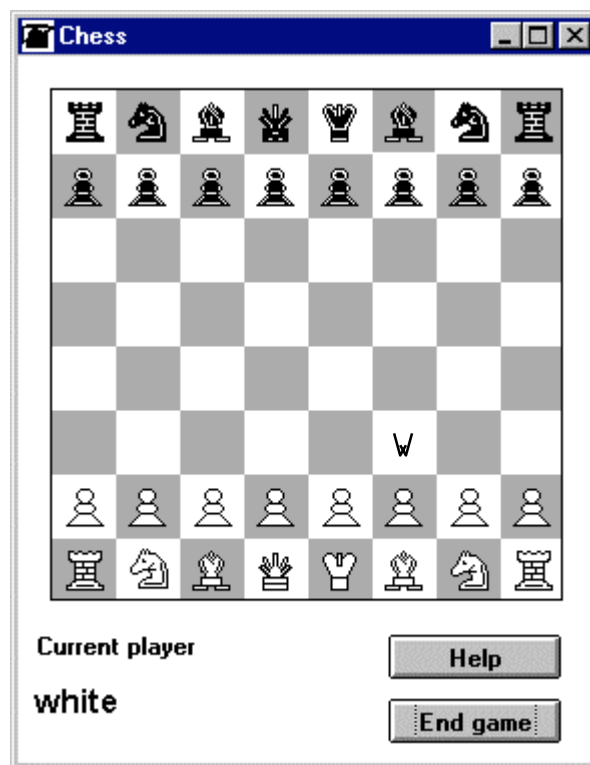


Figure A.3.1. Chess board in starting position and cursor indicating white's turn

Each game begins in the standard configuration shown in Figure A.3.1. A player makes a move by clicking the <select> mouse button over the piece to be moved and then over the desired destination square. Before the first click, the cursor has the shape of a letter indicating which player should move - W for White, B for Black. After the first click, the cursor changes to D (for Destination).

If the move specified by the player is legal, the program displays the move on the chessboard. If the first click identifies an empty square or a square occupied by an opponent's piece, the click is ignored.

If the initial click identifies a legal piece but the destination square is illegal, the attempted move is also ignored. Illegal moves include an attempt to move to a position occupied by one's own piece, moving to a position outside the legal path of the piece or identical to the starting position, and moves along an obscured path (legal knight moves are an exception). A move that exposes one's own king to check is also illegal.

More complicated moves including castling (a move involving a king and a rook), pawn capturing, and pawn replacement upon reaching the opponent's end of the board are not supported. Check mates are recognized only implicitly in that no legal move is possible from a check mate position. Draws are not recognized. When a move results in a check of the opponent's king, the program displays a warning and a red CHECK label. The game ends when a player clicks *End of game*. The *Help* buttons explains the use of the program.

Scenarios.

In keeping with the specification, the term *Player* used in the following scenarios denotes a human player.

Scenario 1: Starting a game between two human players.

1. *Player* executes a statement such as Chess open.
2. *Program* opens board in the state shown in Figure A.3.1, expecting white to make the first move. When the player moves the cursor to the chessboard, it takes the shape of a W indicating that it is the white's turn to play.

Scenario 2. Making a legal move.

1. *Player* clicks one of his or her pieces.
2. *Program* changes cursor to a D, to indicate that the player should click the destination square.
3. *Player* clicks a legal destination square.
4. *Program* redisplay the affected squares.
5. *Program* changes cursor to a W or a B depending on which player moves next.

Scenario 3. Making a legal move resulting in a check.

Same as Scenario 2 but at the end, *Program* displays a dialog warning about the check and the word CHECK in red letters in the game window. The word is removed when the player makes a move that eliminates the check.

Scenario 4. Making an illegal move: First click is an empty square or square occupied by opponent's piece.

1. *Player* clicks an empty square. The shape of the cursor does not change and the click is ignored. The next click is considered to be the first click.

Scenario 5. Making an illegal move: Desired destination is occupied by one's own piece, a piece is blocking the path, the desired move leaves player's king in check, the destination is not on a legal path or is identical to the starting position.

1. *Player* clicks one of his or her pieces.
2. *Program* changes cursor to a D.
3. *Player* clicks a square occupied by his or her own piece.
4. *Program* does not change the chess board and redisplay the initial cursor (B or a W) allowing the player to make a new move.

Scenario 6. End of game by clicking *End of game*.

1. *Player* clicks *End of game*.
2. *Program* displays multiple choice dialog with *Quit*, *New game*, and *Cancel* buttons.
3. *Player* clicks *Quit* and closes the program.

Context Diagram

The major components of the system are the chess board with its pieces and user interface, and the players. The players (including a computer program player) are outside of the scope of the task. The Context Diagram is shown in Figure A.3.2.

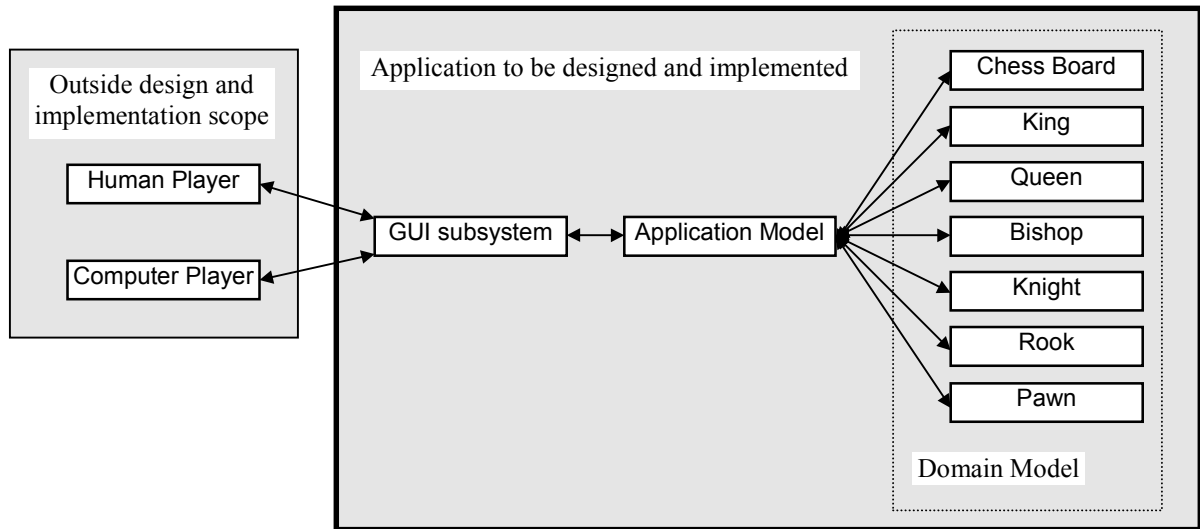


Figure A.3.2. Context Diagram showing components to be designed and components outside the scope of the task.

A.3.2 Preliminary Design

Candidate classes

From the specification, our understanding of chess, and our knowledge of the principles of VisualWorks interfaces we decide on the following candidate classes:

ChessBoard. Domain model of the chess board. Knows the occupant of each square, and which player will move next. Can determine whether a move is legal or not in collaboration with individual pieces.

King. Knows its position and color, helps to determine whether a an attempt to move it is legal. Domain class. Knows its graphical representation.

Queen, Bishop, Knight, Rook, Pawn. Same responsibilities as King but different operation. Domain classes.

ChessGame. Application model defining the window and interfacing the GUI with the domain model.

ChessView. Displays the board and knows how to redraw itself or its parts in response to a new ChessBoard state. A UI class.

ChessController. The controller of ChessView. Captures mouse clicks and converts them to appropriate actions resulting in changes of ChessBoard, and piece positions. Controls cursor image.

Class-level scenarios

After identifying the candidates, we will now examine the scenarios at class-level detail to determine class responsibilities and confirm that we have all the classes we need.

Scenario 1. Starting a game.

1. *Player* executes Chess open.
2. ChessGame initializes its instance variables, opens board in starting state, expecting the first move to be made by white. ChessController displays a W shaped cursor.

Scenario 2. Making a legal move.

1. *Player* clicks one of his or her pieces.
2. ChessController converts cursor coordinates to square row and column, requests confirmation that this is a legal square from ChessBoard, and changes cursor to a D.
3. *Player* clicks a legal destination square.
4. ChessController converts cursor coordinates to square coordinates, asks ChessBoard to confirm and perform the move. ChessBoard performs the necessary tests, asking the selected piece to execute piece-related actions. It asks ChessView to redisplay the current square as empty and display the moved piece in the new position. If the new position was occupied by an opponent's piece, it removes the opponent's piece from the game. ChessGame changes cursor to W or B depending on the player whose turn it is next.

Scenarios 3, 4, and 5 follow the same pattern as Scenario 2 and we leave them as exercises. Scenario 6 does not require any further comments and we can now construct the following preliminary class descriptions:

Preliminary class descriptions

Chess. Application model implementing the main user interface.

Components: chessBoard, chessView

Responsibilities:

- opening - start new game and display user interface
- accessing - chess view
- action methods - buttons
 - help - display help window
 - end of game - display dialog and close if required

Collaborators

ChessBoard, ChessView
chess pieces
ChessView

ChessBoard. Domain model of the chess board.

Components: squares (TwoDList), currentPlayer (Symbol)

Responsibilities:

- initialization
- accessing - pieces, view (via dependency)
- making a move and alternating players
- testing legality of moves and checks

Collaborators

pieces
pieces
pieces

ChessController. Captures mouse clicks, controls cursor.

Components: firstClick (Boolean), currentCursor, blackCursor, whiteCursor, destCursor

Responsibilities:

- initialization - define cursor shapes and control current cursor
- events - tracking mouse position and button action
- control of cursor shape

Collaborators

ChessView. Display of the chess board.

Components:

Responsibilities:

- displaying chessboard with pieces
- updating - response to ChessBoard changes via dependency
- controller access

Collaborators

ChessBoard and pieces
ChessBoard
ChessController

Bishop, King, Knight, Rook, Pawn, Queen - represent individual pieces.

Components: color, position

Responsibilities:

- testing - is proposed move legal?
- visual representation – image of black and white pieces on both backgrounds

Collaborators

From this specification, we can draw the preliminary Object Model in Figure A.3.3.

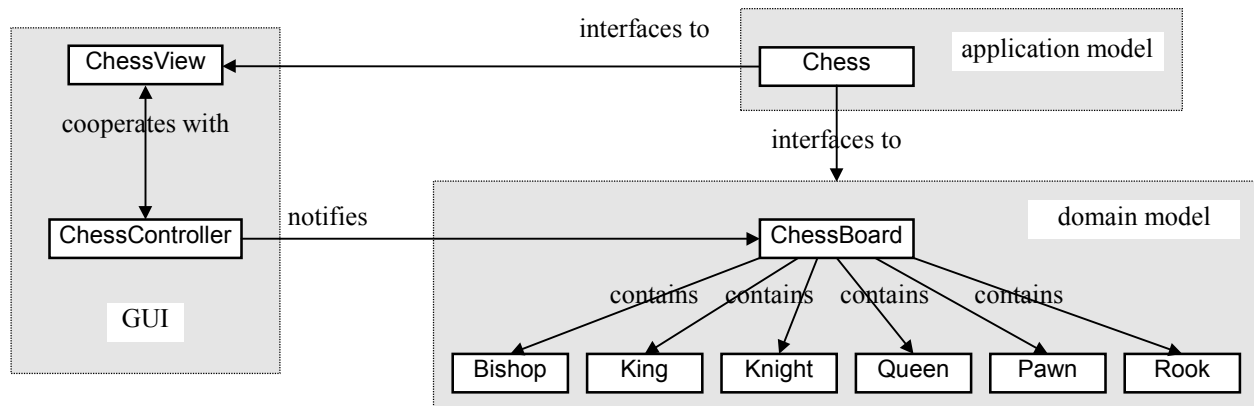


Figure A.3.3. Preliminary Object Model.

A.3.3 Design Refinement

We will now identify abstract classes and the superclass of each class, create a Class Hierarchy diagram, refine responsibilities, and finalize the Object Model.

Abstract classes

The different pieces share a lot of information and behavior: Each has a color and a position on the chess board, all participate in the testing of the legality of a move, and all have their visual representation. We will thus create an abstract class called Piece to define all the common behavior. Class Piece itself is unique and its superclass will thus be Object.

Classes ChessView and ChessController are, of course, in the View and Controller hierarchies. ChessView will be a subclass of View, and ChessController will be a subclass of Controller because we need event notification but no menu.

Class ChessBoard treats ChessView as its dependent and its superclass is thus Model.

Results of this analysis are summarized in the Class Hierarchy diagram in Figure A.3.4 and the following table.

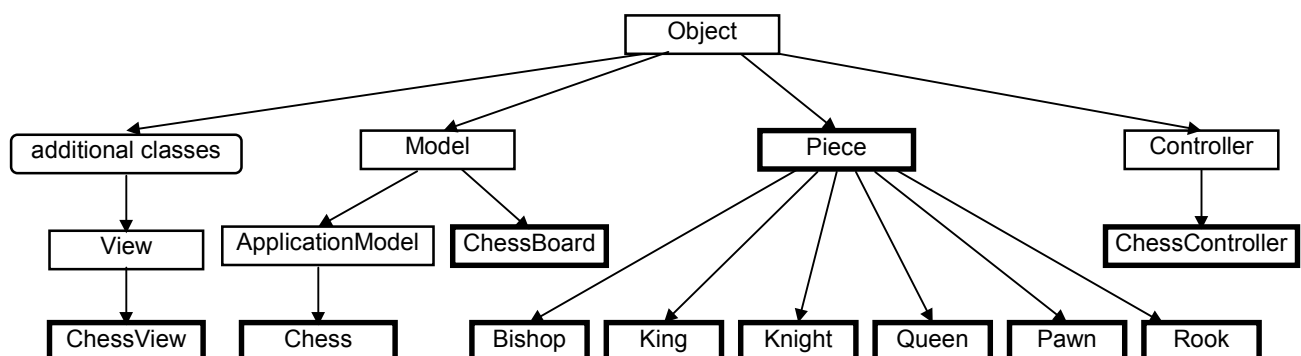


Figure A.3.4. Class Hierarchy. Classes in heavy rectangles are to be implemented.

Class Characteristics:

Class	Type	Superclass	Comment on the choice of superclass
Chess	concrete	ApplicationModel	Mediator between domain model (board and pieces) and GUI.
ChessBoard	concrete	Model	We need dependency to control ChessView.
ChessController	concrete	Controller	Controller provides behaviors such as events, and access to view.
ChessView	concrete	View	View provides persistence and dependency mechanism.
Piece	<i>abstract</i>	Object	There is no suitable superclass in the hierarchy.
King	concrete	Piece	Piece was created as an abstraction of individual pieces.
Queen	concrete	Piece	Same comment.
Bishop	concrete	Piece	Same comment.
Knight	concrete	Piece	Same comment.
Rook	concrete	Piece	Same comment.
Pawn	concrete	Piece	Same comment.

Details of selected behaviors

We will now explore and describe, class by class, the more complicated behaviors in order to obtain sufficiently detailed class descriptions suitable for implementation.

Class Chess

The application opens in the usual way and executes the initialization method. Initialization must create a new ChessBoard and a new ChessView, and assign the ChessBoard to the view as its model. Class ChessBoard must be able to modify information such as the name of current player and this requires that the creation method of Chess provide Chess with access to ChessBoard.

The essence of the program is moving pieces. Every move is initiated by a mouse click, mouse clicks are intercepted by the controller, and we will thus continue with the controller.

Class ChessController

The controller responds to button clicks and controls the cursor via methods with predefined event selectors `redButtonActivity:`, `enterEvent:`, and `exitEvent:`. Message `redButtonActivity:` is sent when the user clicks the red button to select a piece or a destination and its implementation will be as follows:

- Calculate square in row/column coordinates. This is needed to access a piece in ChessBoard.
- If this is the first click of the current player then
 - ask ChessBoard whether the square is a legal starting point
 - remember that the next click will be the second click
 - change cursor shape to D
- If this is not the first click then
 - notify ChessBoard of the destination square
 - ask ChessBoard to make move if it is legal
 - remember that next move will be the first move (even if the move was illegal)
 - if the move was legal, change current player and change cursor to W or B

We will now follow the task of moving a piece by exploring ChessBoard's role in this process. We will return to the controller later.

Class ChessBoard

ChessBoard is responsible for initialization, tests and for triggering the display of the new state.

- Variable `squares` is initialized to empty squares or to appropriate pieces. The rest is obvious.
- The test of whether the starting position is legal checks that the square contains a piece belonging to the current player. If it does not, the move is illegal.
- Our implementation of the test of the destination and the subsequent move will be as follows:
 - Check if the destination square is legal (empty square or square with opponent's piece)
 - Check if the piece can reach the destination. The move is possible if
 - the piece can reach the destination without considering possible obstructions
 - the path from start to destination is not obstructed (except for the knight)
 - Move piece tentatively to the new position without redrawing the board
 - Check whether the move uncovers current player's king. (To do this, opponent's pieces check whether they can reach the king.) If everything is OK, display new position using dependency of `ChessView` on `ChessBoard`, otherwise ignore the move.
 - Is opponent's king in check (player's pieces test whether they can reach opponent's king)? If so, display warning and red label.

To make the operation and design more efficient, we will add the following behaviors and instance variables:

- A method to check whether a move is horizontal, vertical, diagonal, or other ('illegal') because one or more of these tests are required by most pieces. The result is stored in variable `kindOfMove`.
- Instance variables `kings` for quick access to the king piece for tests of checks, and `blackPieces` and `whitePieces` - again to speed up tests of checks.

Class ChessController

We can now return to `ChessController` and describe the details of its responsibilities.

- Initialization (occurs automatically when the controller is created): `firstClick` becomes true, cursor becomes the W cursor because the white player moves first.
- Response to `enterEvent` which is sent when the cursor enters this controller's view: Remember old cursor (to be displayed when the cursor leaves `ChessView`) and change cursor to its proper value.
- Response to `exitEvent` which is sent when the cursor leaves `ChessView`: Reset cursor to what it was before the cursor entered the chess view.

We conclude that the controller must remember the current and old cursors, and the three cursor images (W, B, and D). To construct them, we will use the *Image Editor* tool, remembering to keep the image to [16@16](#) pixels, the prescribed `Cursor` extent.

Class ChessView

Class `ChessView` is responsible for drawing the chess board at the beginning of a new game, updating it after a legal move, and redrawing it after damage. As we know, this is implemented by `displayOn:` and `update`.

Method `displayOn:` goes through all 64 squares of the chess board. It displays a black or a white square when the square is empty, and if the square contains a piece, asks the piece for the appropriate image and displays it.

Updating is invoked after a move, when `ChessBoard` issues a `changed` message. Since each move in this pilot implementation changes only the origin and the destination, we will restrict the response to an update of just those two squares. To be able to do this, we will use `update:with:` and supply the two squares as an `Array` argument of the `with:` keyword.

Class Piece

This class defines shared behaviors and variables of concrete pieces such as King and Queen. The responsibility for testing whether a move can be made is left to concrete subclasses because each concrete subclass uses a different rule. Responsibility for accessing is implemented here.

This completes our design except for the concrete pieces. For a Bishop, legal moves are diagonal. For a Rook, a legal move is either vertical or horizontal. For a Queen, any move that is not classified as 'illegal' is legal; this includes horizontal, vertical, and diagonal moves. Legal moves for Pawn and Knight require special treatment.

The image of a piece is implemented as a class 'resource' method compiled by the Image Editor. These cached images will be used as required.

We will now update our class descriptions, restricting ourselves to a new listing of variables. We don't preclude the possibility that we may have to add auxiliary instance variables during implementation to improve efficiency or to cover any gaps that may still be present.

Updated class descriptions

Class Chess

Instance variables: chessBoard, chessView.

Class ChessBoard

Instance variables: squares, kings, kindOfMove, blackPieces, whitePieces, currentPlayer, oldPosition, newPosition, removedPiece (needed when restoring a tentatively moved piece when the move is illegal).

Class ChessController

Instance variables: firstClick, oldCursor, currentCursor, blackCursor, destCursor, whiteCursor.

Class ChessView

Instance variables: none.

Class Piece

Instance variables: color, position.

Class Bishop and other concrete pieces

Instance variables: none.

Object Model diagram

To complete design, we update the Object Model diagram as in Figure A.3.5.

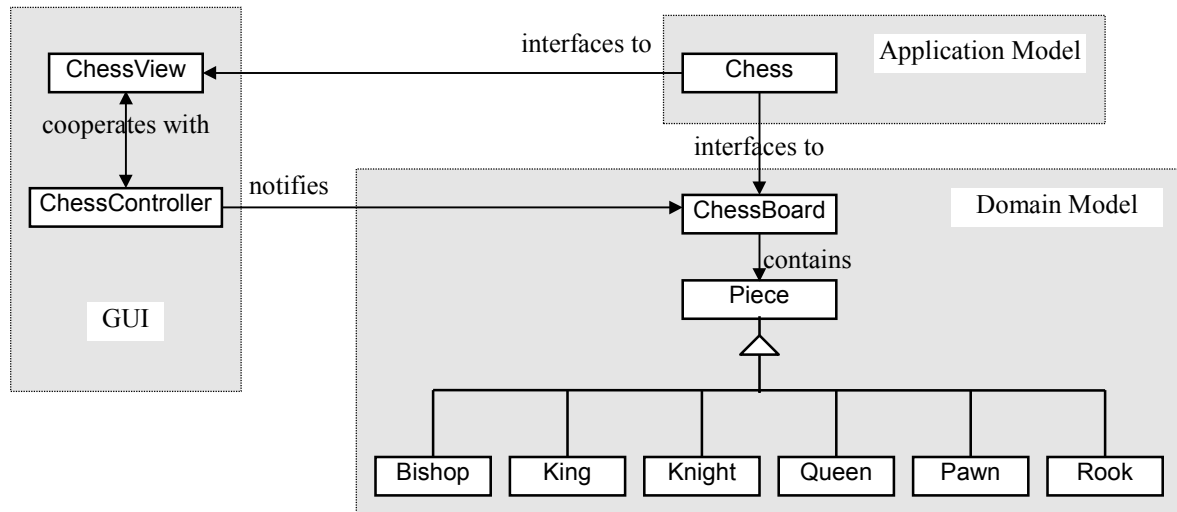


Figure A.3.5. Object Model diagram of chess game classes.

A.3.4 Implementation

Most of the implementation is straightforward and we will limit ourselves to the less trivial methods. The presentation will be organized by class, and classes will be listed alphabetically.

Class Bishop

The creation message for all pieces is `color: aColorValue position: aPoint` and is defined in class `Piece`. Besides the resource methods that define bishop's image, the only protocol defined in this class is testing of a legal move. The test checks that the requested move is diagonal as follows:

legalMove: kindOfMove to: aPoint

"Return true if this piece can move to this destination."
^kindOfMove == #diagonal

where `kindOfMove` is a `Symbol` calculated by `ChessBoard` when the player makes a move. As you can see, its use considerably simplifies the test. The argument `aPoint` is necessary for pieces such as the knight and since the `legalMove:to:` message is used polymorphically by all pieces, both arguments must be included even when they are not always needed.

Class Chess

is the application model and its opening message initializes the program as follows:

initialize

```

chessBoard := ChessBoard newFrom: self. "Let ChessBoard know about myself for accessing."
chessView := ChessView new initialize.
chessView model: chessBoard

```

After a player makes a move, `ChessBoard` switches players and sends the following message to change the player label in the window.

player: aPlayer

"Players switched. Assign player and display color label on the UI, hiding the other label."
| otherPlayer |

```
otherPlayer := aPlayer == #white
    ifTrue: [#black]
    ifFalse: [#white].
(self builder componentAt: otherPlayer) beInvisible.
(self builder componentAt: aPlayer) beVisible
```

The operation is based on our decision to paint all possible labels on the canvas and display and hide them selectively as needed.

Class ChessBoard

Holds board information and needs to know about Chess to notify it to update display after a move has been made.

initializeFrom: app

```
"Initialize everything for the first game."
appModel := app.
self newGame
```

The lengthy code that initializes the squares and creates pieces is taken out into the following methods:

newGame

```
"Initialize square information and create white and black pieces."
self initializeSquares.
self initializePieces.
currentPlayer := #white
```

Method newGame: is reused as a response to the selection *New game* in the closing dialog of the game. The details of initialization are as follows:

initializeSquares

```
"Create holder for board description and assign a piece to each square."
| pos |
squares := TwoDList columns: 8 rows: 8.
1 to: 8 do: [:col | "Blank empty squares."
    1 to: 8 do: [:row | squares at: row @ col put: nil]].
1 to: 8 do: [:col | "Place remaining pieces."
    "Place pawns."
    squares at: (pos := col @ 2) put: (Pawn color: #black position: pos).
    squares at: (pos := col @ 7) put: (Pawn color: #white position: pos)].
    "Place all remaining pieces."
    squares at: (pos := 1 @ 1) put: (Rook color: #black position: pos).
    squares at: (pos := 1 @ 8) put: (Rook color: #white position: pos).
    squares at: (pos := 8 @ 1) put: (Rook color: #black position: pos).
    squares at: (pos := 8 @ 8) put: (Rook color: #white position: pos).
    squares at: (pos := 2 @ 1) put: (Knight color: #black position: pos).
    squares at: (pos := 2 @ 8) put: (Knight color: #white position: pos).
    squares at: (pos := 7 @ 1) put: (Knight color: #black position: pos).
    squares at: (pos := 7 @ 8) put: (Knight color: #white position: pos).
    squares at: (pos := 3 @ 1) put: (Bishop color: #black position: pos).
    squares at: (pos := 3 @ 8) put: (Bishop color: #white position: pos).
    squares at: (pos := 6 @ 1) put: (Bishop color: #black position: pos).
    squares at: (pos := 6 @ 8) put: (Bishop color: #white position: pos).
    squares at: (pos := 4 @ 1) put: (Queen color: #black position: pos).
    squares at: (pos := 4 @ 8) put: (Queen color: #white position: pos).
    squares at: (pos := 5 @ 1) put: (King color: #black position: pos).
    squares at: (pos := 5 @ 8) put: (King color: #white position: pos)
```

Auxiliary variables holding white pieces, black pieces, and kings are initialized as follows:

initializePieces

"Initialize variables holding black pieces, white pieces, and kings using information about squares."

```
whitePieces := OrderedCollection new: 16.  
blackPieces := OrderedCollection new: 16.  
kings := Array new: 2.  
squares do: [:piece | piece isNil  
    ifFalse: [piece color == #white  
        ifTrue: [whitePieces add: piece.  
            piece isKing ifTrue: [kings at: 1 put: piece]]  
        ifFalse: [blackPieces add: piece.  
            piece isKing ifTrue: [kings at: 2 put: piece]]]]]
```

The essence of the program is the handling of individual moves and besides the necessary manipulation of the user interface, this requires various checks. The method is implemented as follows:

move

"Attempt to make a move and return true if the move is legal. Do nothing and return false otherwise."

```
| piece destPiece |  
piece := squares at: oldPosition. "Piece at first click position."  
"Move is legal only if destination is empty or occupied by opponent."  
((destPiece := squares at: newPosition) isNil or: [destPiece color ~= currentPlayer])  
    ifFalse: [^false].  
"Can this piece go to the specified destination?"  
(self piece: piece canReach: newPosition from: oldPosition)  
    ifFalse: [^false].  
"Make the move tentatively."  
self move: piece to: newPosition.  
"Does the move leave the player's king exposed? If so, retract."  
self inCheck ifTrue: [self restore: piece. ^false].  
(appModel builder componentAt: #check) beInvisible.  
"Move is OK, get it displayed via dependency."  
self changed: #position with: (Array with: oldPosition with: newPosition).  
"Does the move put opponent's king in check? Display red label if true."  
self checks ifTrue: [(appModel builder componentAt: #check) beVisible].  
"Move completed, get ready for next player's move."  
self switchPlayer.  
^true
```

Although this method works correctly, its design is unsatisfactory. In particular, ChessBoard is a domain object and its implementation should be independent of the user interface. Yet, this method directly accesses a specific component of the user interface – the check label. If the interface changed, this reference would become invalid, and if we wanted to use ChessBoard in another application, we couldn't. The proper way for a domain object to communicate with the UI is by notifying the application model of the change that occurred and leave the implementation details to it. We leave this modification as an exercise.

As you have noticed, the move method performs a variety of tests and some of them were delegated to the following auxiliary methods:

piece: aPiece canReach: point2 from: point1

"Can aPiece go from point1 to point2?"

```
"Determine the kind of move (such as horizontal or vertical) being attempted."  
self kindOfMoveFrom: point1 to: point2.  
"Ask the piece if this move is OK."  
(aPiece legalMove: kindOfMove to: point2)  
    ifFalse: [^false].  
"Move is OK but is the path clear?"  
^self isClearFrom: point1 to: point2
```

In here, method legalMoveFrom:to: checks whether aPiece can, in principle, move from point1 to point2. It does not check whether the path is clear. The method is defined differently for each piece because

it depends on how the piece moves. We saw how Bishop treats `legalMoveFrom:to:` and other pieces define it similarly. Message `isClearFrom:to:` is sent only after confirmation that the move is, in principle, legal; it checks whether the path between the two points is clear and its definition is as follows:

isClearFrom: point1 to: point2

"Return true if the path from point1 to point2 is clear. Delegate the problem to specialized direction-dependent methods."

```
kindOfMove == #horizontal ifTrue: [^self horizontalClearFrom: point1 to: point2].
kindOfMove == #vertical ifTrue: [^self verticalClearFrom: point1 to: point2].
kindOfMove == #diagonal ifTrue: [^self diagonalClearFrom: point1 to: point2].
^true "Knight does not care about obstructions."
```

The method uses specialized methods that test a horizontal path, a vertical path, and a diagonal path. We could reduce the number of tests somewhat but the definition would be less readable and we thus leave it in this form. The following is an implementation of one of these test methods:

horizontalClearFrom: point1 to: point2

"Return true if the horizontal line between the two points is clear."

```
| x1 x2 y step |
x1 := point1 x.
x2 := point2 x.
y := point1 y.
x1 > x2 ifTrue: [step := -1]
        ifFalse: [step := 1].
x1 + step to: x2 - step by: step do: [:x | (squares at: x @ y) isNil ifFalse: [^false]].
^true
```

The method that determines the kind of move is defined as follows:

kindOfMoveFrom: point1 to: point2

"Is the move horizontal, vertical, diagonal, or 'illegal'? Return appropriate symbol."

```
kindOfMove := point1 y = point2 y
              ifTrue: [#horizontal]
              ifFalse: [point1 x = point2 x
                        ifTrue: [#vertical]
                        ifFalse: [(point1 x - point2 x) abs = (point1 y - point2 y) abs
                                ifTrue: [#diagonal]
                                ifFalse: [#illegal]]]
```

The chessboard knows about all pieces and so we leave it in charge of king checks. One such check requires finding whether a piece in a new position checks the opponent's king, the other checks whether it uncovers the player's own king to check. Both tests work on the same principle. As an example, the test to determine whether a move leaves the player's king in check is defined as follows:

inCheck

"Test whether the king of the current player is in check by asking each opponent piece. Return true if check."

```
| position |
position := self myKing position.
"Ask each of the opponent's pieces if it can reach the king."
self opponentPieces do: [:piece | (self
    piece: piece
    canReach: position
    from: piece position)
    ifTrue: [^true]].
^false
```

Finally, the following method restores the chess board to its original position if we must take back a tentative move that resulted in a check to one's own king:

restore: aPiece

"Attempted move is illegal because it exposes my king to check. Restore original state of board domain model. No need to redraw because we have not drawn anything yet."

"Move piece to its original position."

squares at: oldPosition put: aPiece.

"Restore taken piece if any."

removedPiece isNil iffFalse: [self opponentPieces add: aPiece].

squares at: aPiece position put: removedPiece.

aPiece position: oldPosition

Class ChessController

This class is in charge of user interaction for the chess view. Its initialization method prepares the controller for the first click and defines the three cursor shapes:

initialize

firstClick := true.

whiteCursor := Cursor image: ChessController aWhiteCursor asImage
mask: ChessController aWhiteCursorMask asImage
hotSpot: 4 @ 8
name: 'white moves'.

blackCursor := Cursor image: ChessController aBlackCursor asImage
mask: ChessController aBlackCursorMask asImage
hotSpot: 4 @ 8
name: 'black moves'.

destCursor := Cursor image: ChessController aDestCursor asImage
mask: ChessController aDestCursorMask asImage
hotSpot: 4 @ 8
name: 'destination'.

currentCursor := whiteCursor

The 'resource' class methods aWhiteCursor, aBlackCursor, and aDestCursor supply the cursor. They were generated automatically by painting the image and mask shapes by the *Image Editor*. Note that the image and the mask are stored by two different commands (Figure A.3.6).

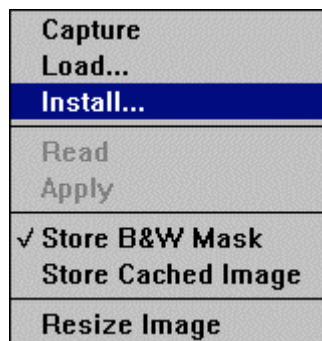


Figure A.3.6. Image Editor allows installing a shape as a cached image or as a mask.

Cursor control methods are implemented by the predefined event selectors as follows:

enterEvent: event

"Remember cursor on entry and switch to the cursor appropriate for the current state."

oldCursor := Cursor currentCursor.

Cursor currentCursor: currentCursor.

currentCursor show

exitEvent: event

"Return to the saved cursor."

Cursor currentCursor: oldCursor.
Cursor normal show

The calculation of currentCursor occurs when the mouse button is clicked and is explained below.

The activity associated with pressing the <select> button when the player selects a piece or a destination again uses a predefined event selector. It is defined as follows:

redButtonPressedEvent: event

```
"User clicked <select> button over a square. Relate this to the move now in progress."  
  | square |  
  "Convert cursor coordinates to row and column numbers."  
  square := ((self sensor cursorPointFor: event) / self view squareExtent) truncated + (1 @ 1).  
  firstClick ifTrue: "First click selects piece. Check legality, notify ChessBoard, change cursor."  
    [(self model isSquareLegal: square)  
     ifFalse: [^self].  
     self model oldPosition: square. "Remember old position to be able to restore it."  
     firstClick := false.  
     currentCursor := destCursor]  
  ifFalse: "Second click selects destination. Check, make move, change cursor."  
    [self model newPosition: square.  
     self model move.  
     firstClick := true.  
     currentCursor := self model currentPlayer == #white  
       ifTrue: [whiteCursor]  
       ifFalse: [blackCursor]].  
  "Ask new cursor to display itself."  
  currentCursor show
```

Class ChessView

The main responsibility of this class is, of course, displaying. The displayOn: method is automatically executed for every damage and change, and during opening. It must be able to redraw the whole board and we implement it by enumerating over all squares, displaying squares for the empty ones, and asking occupying pieces to supply their image for the given background (explained later). The definition is as follows:

displayOn: aGraphicsContext

```
"Draw the board for the current configuration of pieces."  
squareExtent := self squareExtent. "Get the extent of a square in pixels."  
1 to: 8 do: [:row | 1 to: 8  
  do: [:column |  
    | background piece square |  
    "Calculate view coordinates of square from its row and column numbers."  
    square := column - 1 * squareExtent @ (row - 1 * squareExtent ).  
    "Calculate background color."  
    background := (row + column) even  
      ifTrue: [#white]  
      ifFalse: [#lightGray].  
    piece := self model squares at: column @ row. "Piece on current square, if any."  
    piece isNil  
      ifTrue: "The square is empty."  
        [aGraphicsContext paint: (ColorValue perform: background);  
         displayRectangle: (0 @ 0 extent: squareExtent @ squareExtent )  
           at: square]  
      ifFalse: "The square contains a piece. Get its image from the piece."  
        [aGraphicsContext displayImage:  
          (piece imageForPlayer: piece color  
           onBackground: background)  
          at: square]]]
```

Note that we added a new instance variable called `squareExtent` to hold the size of a square. Its value will be useful for updating as well and `Chess` calculates it from the actual window size using lazy evaluation because during initialization, the chess board view has not yet been built. Alternatively, we could do the calculation in `postBuildWith:`.

Another responsibility of the view is response to update requests. The `update` method is executed whenever `ChessBoard` (the model of `ChessView`) executes the `changed` message which happens when it accepts a legal move. We use the `update:with:` form because the model (a `ChessBoard`) must supply the two `Rectangle` objects corresponding to the squares that are to be redrawn. The implementation is as follows:

update: aSymbol with: anArray

"The model has changed. Invalidate the affected squares."

```
| square |
1 to: 2 do: [:i | square := anArray at: i.
self invalidateRectangle: (square - 1 * squareExtent extent: squareExtent)]
```

Note that we restrict invalidation to the two affected squares. Although this does not speed up redrawing, it eliminates flashing which would occur if the whole board was redrawn.

Class Knight

Like all other pieces, `Knight` only defines its shape and a test for a legal move. The definition is as follows:

legalMove: kindOfMove to: aPoint

"Return true if this piece can move to aPoint. Two vertical steps and one horizontal step or vice versa are legal."

```
| absX absY |
absX := (position x - aPoint x) abs.
absY := (position y - aPoint y) abs.
kindOfMove == #illegal    "#horizontal, #vertical, and #diagonal, are disallowed for knights."
ifTrue: [^(absX = 2 and: [absY = 1])
        or: [absX = 1 and: [absY = 2]]].
^false
```

Class Piece

This class is used mainly for shared variables and does not define any interesting methods, except for the creation method

color: aSymbol position: aPoint

```
^(self new) color: aSymbol; position: aPoint
```

calculation of the piece image to display for the current player

imageForPlayer: symbol1 onBackground: symbol2

```
^(self class imageForPlayer: symbol1 onBackground: symbol2) asImage
```

where

imageForPlayer: symbol1 onBackground: symbol2

"Retrieve the appropriate image for this piece and for the current player."

```
^self perform: (symbol1 == #white
    ifTrue: [symbol2 == #white
        ifTrue: [#whiteOnWhite]
        ifFalse: [#whiteOnBlack]]
    ifFalse: [symbol2 == #white
        ifTrue: [#blackOnWhite]
        ifFalse: [#blackOnBlack]])
```


and method `isKing` which defines every piece *not* to be a King by default. (Method `isKing` is used to assemble the array of kings.) The definition is listed below and only class `King` redefines it to return `true`. This is a typical trick used to eliminate multiple definitions when only one class has a different implementation, and method `isNil` is an example of a library method defined in this way.

`isKing`
`^false`

These selected methods should give you a good idea of our implementation and we leave the rest and improvements to you as exercises.

Main lessons learned:

- A cursor may be constructed with the Image Editor. Remember that cursor size is limited to 16@16 and that the mask shape must be stored as a mask.
- Redrawing should always be restricted to a minimum to minimize flashing. This can be achieved by using `invalidateRectangle:` instead of `invalidate`. In some situations, one can also design `displayOn:` to redraw only the required part of the view.
- Smalltalk normally leaves damage repair by invalidation until the processor has nothing more important to do (lazy invalidation). To redisplay without delay, use `invalidateRectangle:repairNow:`.

Exercises

1. Complete and test the implementation.
2. Redesign the application by introducing a new class called `Move` with components `kindOfMove` (a `Symbol`) and `startSquare` and `endSquare` (both `Points`). Consider implementing squares as instances of `BoardSquare` capable of displaying themselves and knowing their chess piece.
3. Implement all rules of chess such as castling, taking en passant, and so on. Does this extension require a major change in the original design?
4. We have seen that illegal situations such as division by zero are generally handled by the exception mechanism. Give a detailed description of how illegal moves could be handled as exceptions.
5. Remove the direct coupling between the domain model `ChessBoard` and the user interface.
6. Moving a piece takes a noticeable time in the current implementation. When you analyze what is happening, you will find that although we limit display to only two squares (using `invalidateRectangle:`), the whole `displayOn:` method must still be executed and all 64 squares recalculated. It seems that performance could be improved by making sure that only the two affected pieces are re-evaluated. Estimate how much this modification could speed up redisplay and implement it.
7. Measure how long it takes to draw the interface for various drawing strategies including `invalidate`, `invalidateRectangle:`, and the improvement suggested in the previous exercise. Compare the seriousness of flashing of the various approaches.
8. How would our implementation have to be modified if we wanted to use it as the interface of a computer player?
9. Improve the game by allowing players to initiate an automatic log of their moves. Allow the game log to be printed or replayed under the control of an additional button. Keep names and scores of players.
10. Enforce a time limit on each move. When a move is not completed within the limit, the player automatically loses the game.
11. Write a series of simple computer chess player programs starting with a program that makes random moves.
12. Use or modify suitable parts of the chess program to answer the following question:
 - i. Which positions can be reached by a knight from a given starting position in exactly three moves?
 - ii. Which positions can be reach by a knight from a given starting position in at most three moves?

- iii. How many moves are required for a knight to reach a given square from a given starting square?
- iv. What is the maximum number of moves for a knight to reach any position on the board from a given starting square?
- v. Find a configuration of eight queens such that none can reach any other in one move. This problem is called the problem of eight queens.
- vi. Find all queen positions for the previous exercise.

Conclusion

In this appendix, we presented the complete development of an application with a custom user interface component. The notable features were the creation of custom cursors and handling of input events. We also used view invalidation restricted to a small part of the view and mentioned that although this does not speed up display, it minimizes flashing and thus improves the quality of the user interface.

Appendix 4 - Classes, Metaclasses, and Metaprogramming

Overview

This chapter examines the ultimate question - the nature of classes. As we know, everything in Smalltalk is an instance of a class and this applies even to classes themselves. We will see that every class is an instance of a metaclass and that the hierarchy of metaclasses parallels the hierarchy of classes. Metaclasses are, of course, also objects and therefore instances of other, perhaps even more abstract classes. It turns out that the reality is simpler and that each metaclass is an instance of one special class called, rather predictably, *Metaclass*. Since *Metaclass* is a class, it is itself an instance of the metaclass *Metaclass* - according to the above general rule. The metaclass of *Metaclass* is, naturally, an instance of *Metaclass* like all metaclasses. Since the metaclass hierarchy parallels the corresponding class hierarchy, all metaclasses are subclasses of the metaclass of class *Object*. A few additional classes that define the shared behavior of classes complete the picture.

Classes and metaclasses are not only an elegant solution to the problem of making everything an instance of a class. The structure also has some very useful and important implications that are essential for class creation, editing, and gathering of information about classes. Programming based on the class-metaclass structure is called metaprogramming and we will give several examples of its use.

A 4.1 Classes and Metaclasses

All Smalltalk objects are instances of classes: A Smalltalk character is an instance of *Character*, an array is an instance of *Array*, a browser is an instance of *Browser*, and so on. To be fully consistent with this principle, all classes should themselves be instances of other classes - and indeed they are. These higher level classes are called *metaclasses* and their names are identical to the names of the classes themselves. Class *Array* is thus an instance of metaclass *Array*, class *Character* is an instance of metaclass *Character*, and so on. To avoid proliferation of names and confusion, metaclasses don't have separate names and can only be accessed by sending the class message to the class. As an example, to access the metaclass of *Array* execute expression

Array class

When executed with *inspect*, this opens the inspector in Figure A 4.1.

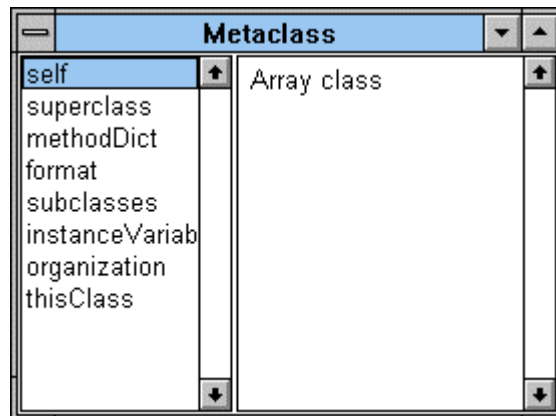


Figure A 4.1. Inspector on Array class - the metaclass of Array.

The hierarchy of metaclasses parallels the hierarchy of classes, and when you create a new class, its metaclass is automatically created with it (Figure A 4.2).

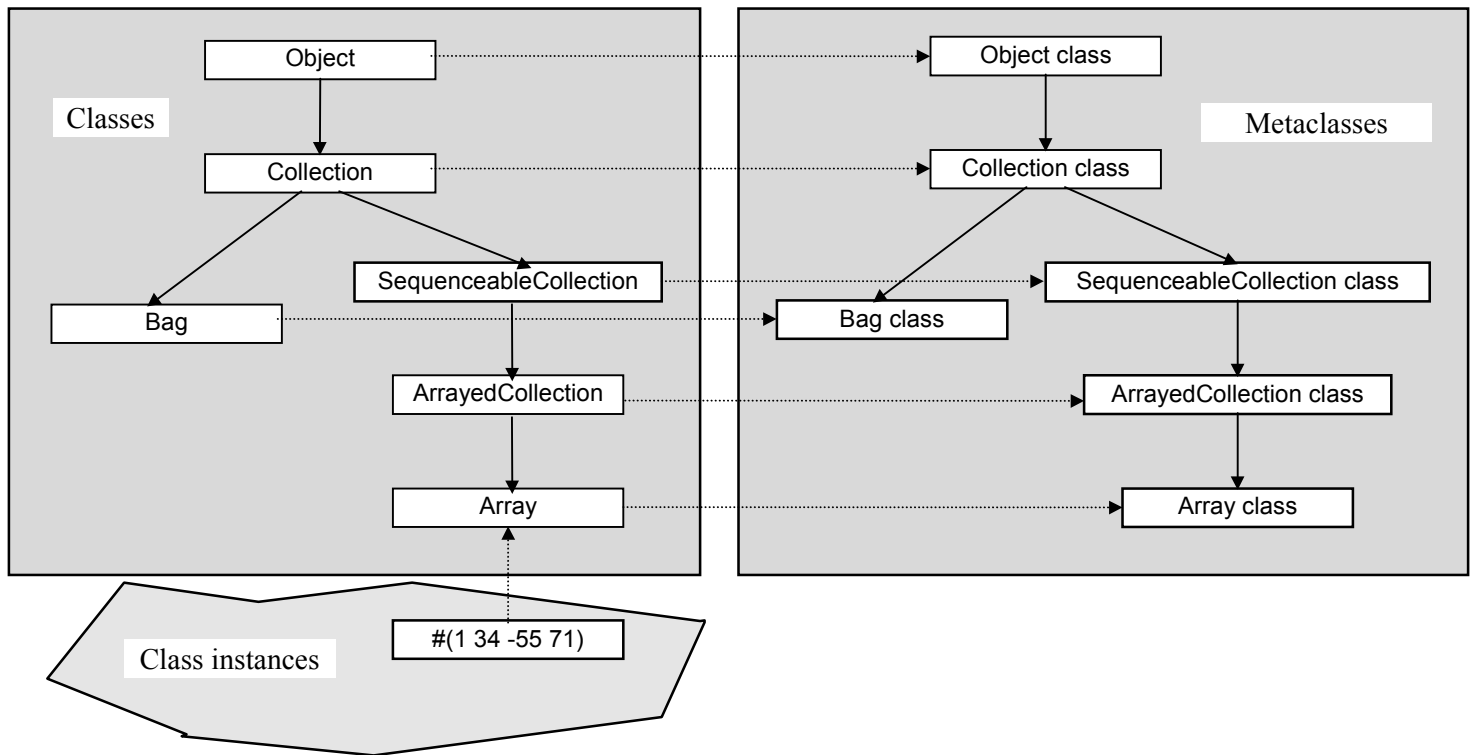


Figure A 4.2. Class hierarchy is paralleled by metaclass hierarchy. The➔ relationship means 'is instance of' whereas —➔ represent subclassing.

These principles immediately invite several questions such as - what is the class of a metaclass, how are metaclasses different from classes, how does the existence of a parallel hierarchy tree relate to the fact that the superclass of every class is class Object, do metaclasses have any methods and instance or class variables, and finally - what is the use of this elaborate structure.

Main lessons learned:

- Every class is a single instance of its metaclass.
- A class and its metaclass have the same name.
- A class is accessed by its name but its metaclass is accessed by sending message `class` to the class.
- Metaclasses are not accessible through the browser but can be accessed by an inspector.
- When a class is created, its metaclass is automatically created with it.
- Metaclasses are arranged in an inheritance tree that parallels the inheritance tree of classes.

Exercises

1. Inspect the components of the metaclass of Character and the metaclass of Array. Make preliminary comments on the variables.

A 4.2. What is the complete class hierarchy?

If an 'ordinary' class is an instance of its metaclass, what is a metaclass? The answer is easy to find - we only need to ask each metaclass what is its class as in

Integer class class "Returns Metaclass"

Array class class "Returns Metaclass"

We conclude that all metaclasses are instances of class Metaclass (Figure A 4.3).

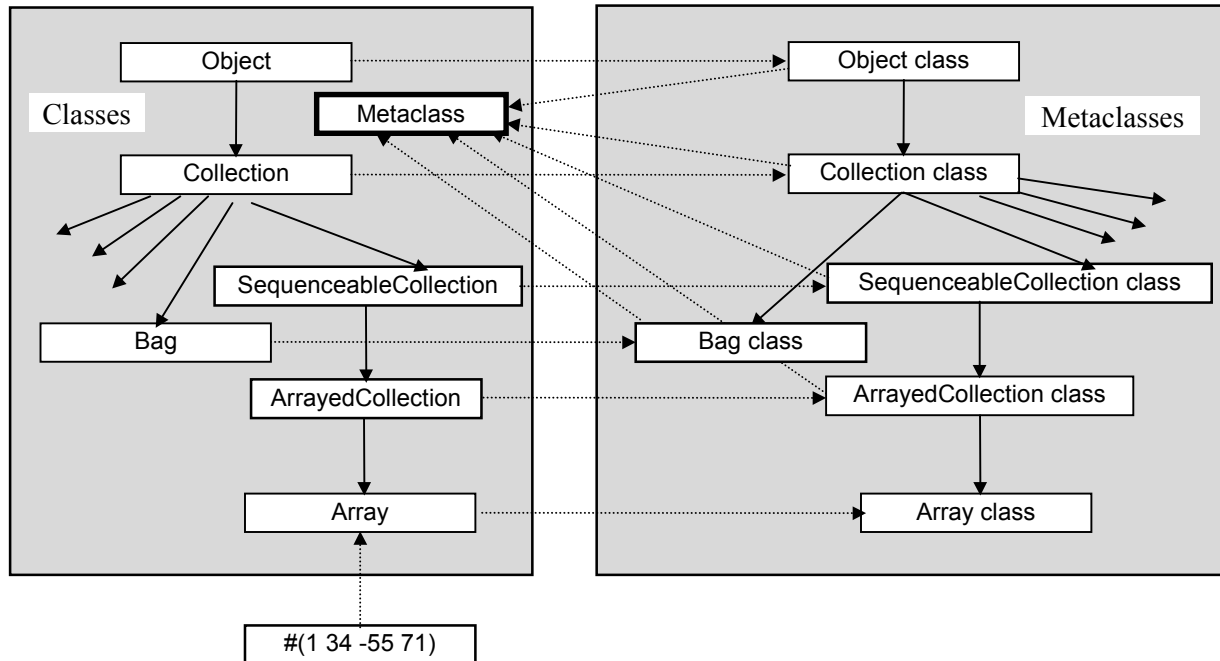


Figure A 4.3. Every metaclass is an instance of class Metaclass.

But if this is so, then what is the class of Metaclass? Since Metaclass is a class just like Array or Integer, the answer, of course, is that Metaclass is an instance of Metaclass class. And what is the metaclass of Metaclass class? The answer is again simple - since Metaclass class is a metaclass and since the metaclass of every class is Metaclass, the metaclass of Metaclass class is Metaclass (Figure A 4.4). This is very natural when you realize that everything is governed by the following two rules:

1. The class of class X is its metaclass X class.
2. Every metaclass is an instance of class Metaclass.

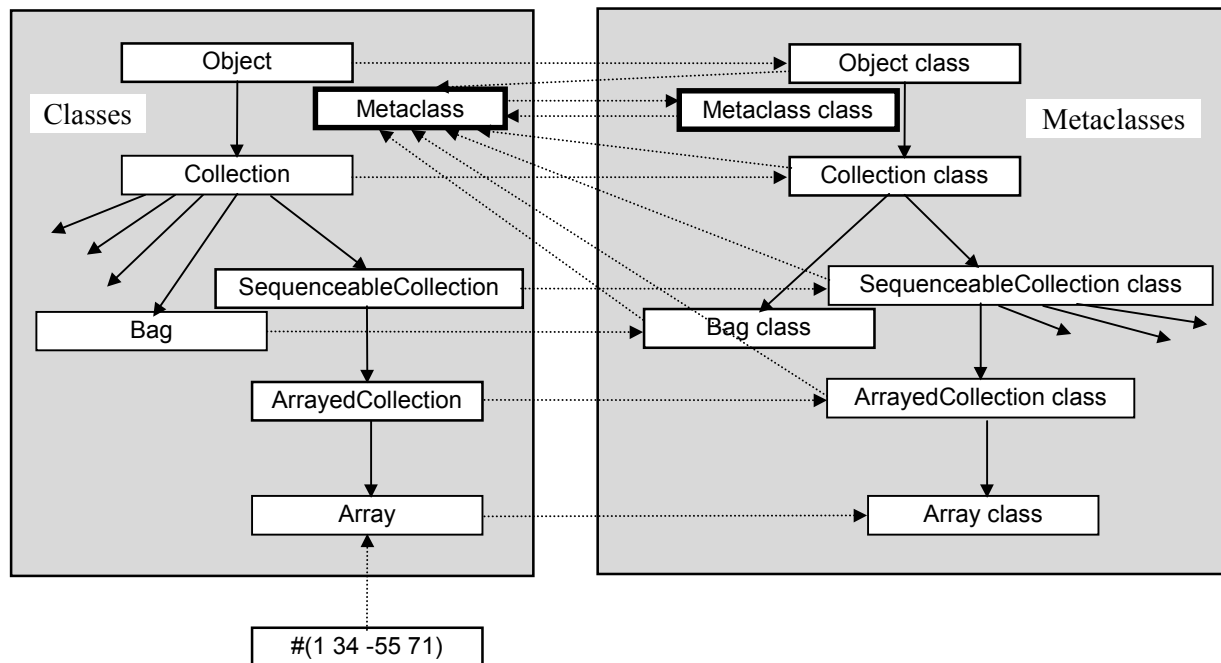


Figure A 4.4. The metaclass of Metaclass is Metaclass class. Metaclass class is an instance of Metaclass.

Up to this point, we have been concentrating on metaclasses and neglected the subclass relationship. As an example, we have not said what is the superclass of Object class - the head of our metaclass hierarchy. To find out, execute

Object class superclass

The answer is that the superclass of Object class is Class (Figure A 4.5). Class Class itself is, of course a subclass of Object - like every other class. And the metaclass of Class is Class class.

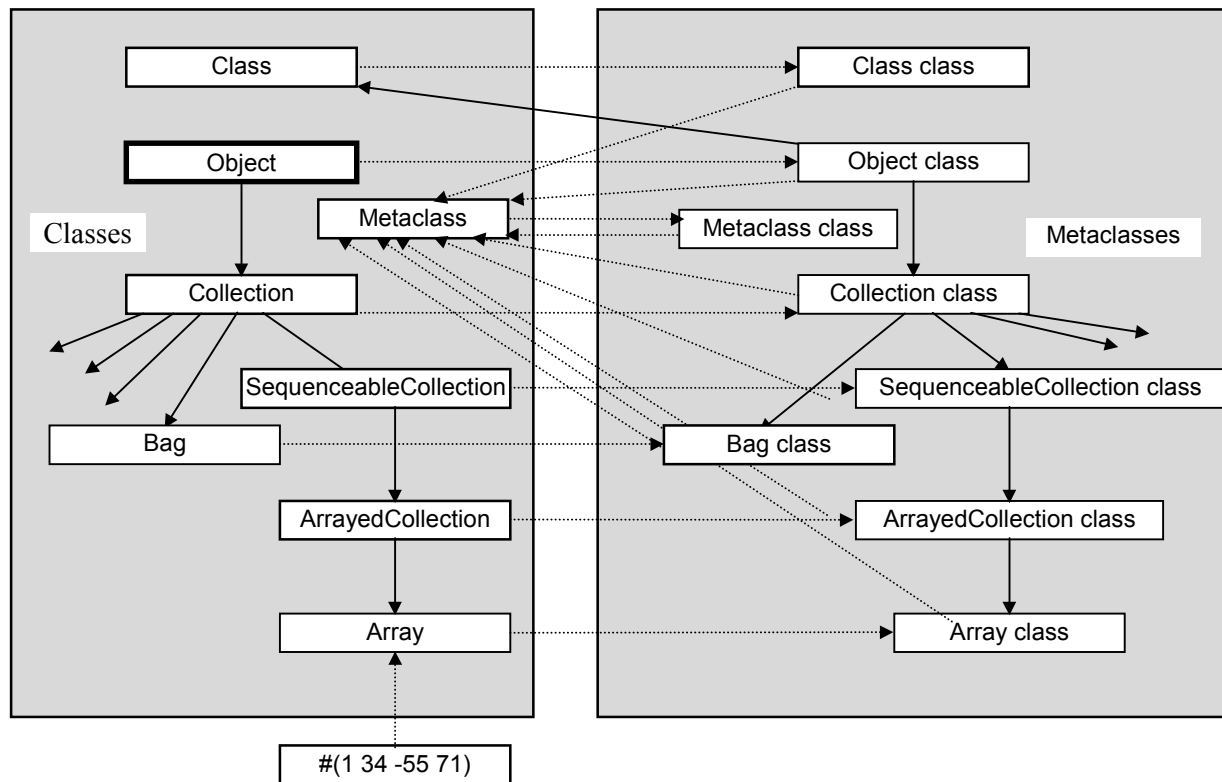


Figure A 4.5. The superclass of metaclass Object class is class Class.

However, our picture is incomplete because there are two additional classes called Behavior and ClassDescription between Class and Object. This last refinement is shown in Figure A 4.6.

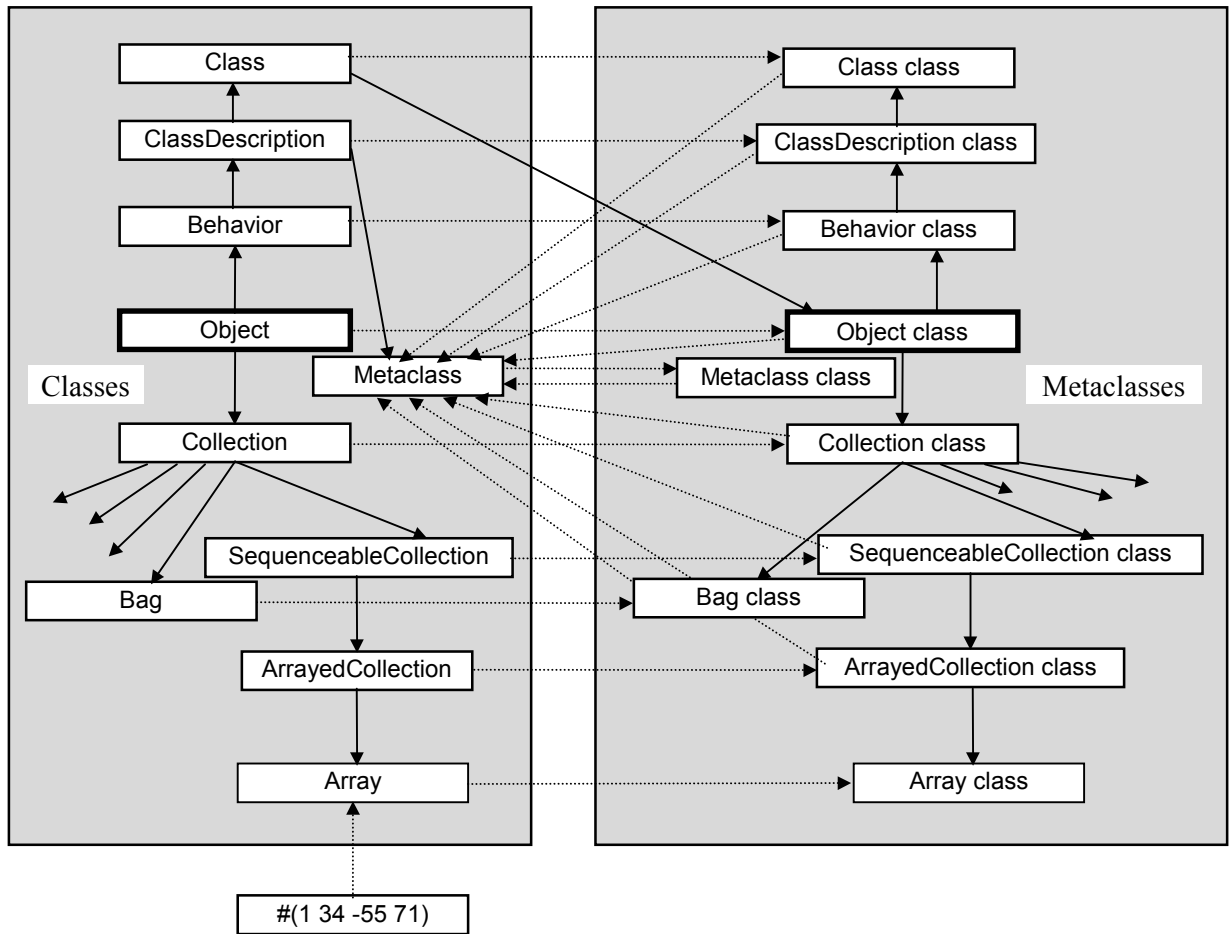


Figure A 4.6. Complete hierarchy of Smalltalk classes. Object is the superclass of all classes and all metaclasses are instances of Metaclass.

Main lessons learned:

- Every class is an instance of its metaclass and every metaclass is an instance of class Metaclass.
- The superclasses of Metaclass are Class, ClassDescription, Behavior, and Object, in this order.
- Object remains at the top of the class hierarchy, including metaclasses.

Exercises

1. Use the System Browser to find all superclasses of metaclass Date class.

A 4.3 What are the main properties of metaclasses?

Unlike most 'ordinary' classes, each metaclass has only one instance - its class. As an example, the only instance of metaclass Array class is class Array. Also, messages that we have so far classified as class messages are, in fact, instance messages of the metaclass. As an example, all class messages of class Array are instance messages of metaclass Array class., and all class messages of Object are instance messages of Object class:

Object *class* selectors asSortedCollection "Returns selectors of all *class* messages of Object."
Object selectors asSortedCollection "Returns selectors of all *instance* messages of Object."

This arrangement is logically related to the general rule of message execution:

When a message is sent to an object, the search for its definition begins in the class of the receiver.

As an example, when we send 13 factorial, the search for the declaration of factorial starts in class SmallInteger, the class of object A 4. It is thus quite consistent that when we send, for example, String new the search for the declaration of new starts in the class of String, in other words, in metaclass String class. And if the declaration of the message is not found there, the search continues in the superclass of String class and so on.

Besides being responsible for class methods, metaclasses inherit all their behavior from their superclasses Class, ClassDescription, Behavior, and Metaclass, All grouped in category Kernel-Classes. We will examine the most important of these shared behaviors starting with the simplest and most general of these classes - Behavior - and then explore the functionality that its subclasses add to it. First, however, here is again the relevant part of the hierarchy:

```
Behavior ('superclass' 'methodDict' 'format' 'subclasses')
  ClassDescription ('instanceVariables' 'organization')
    Class ('name' 'classPool' 'sharedPools')
      ... all metaclasses ...
      Metaclass ('thisClass')
```

and its graphical representation in Figure A 4.7.

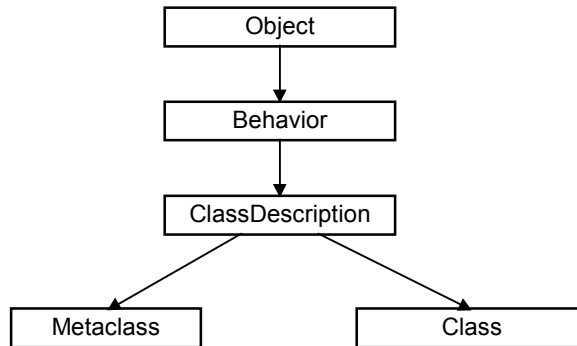


Figure A 4.7. Metaclass-defining classes.

Main lessons learned:

- Class methods of a class are instance methods of its metaclass.

Exercises

1. We have seen that class messages of a class are instance messages of its metaclass. What is the relationship of a class, its metaclass, and instance, class, and class instance variables?

A 4.4 Class Behavior

Class Behavior defines most of the functionality shared by classes and metaclasses. It is responsible for the basic instance creation protocol, the protocol that finds the subclasses and superclasses of a class, access to methods defined in a class, and the kind of the class (more on this later). It also provides the mechanism for finding all instances of a class. Class Behavior does *not* have information that would allow a class to know about the names of its instance variables and message protocols, its class comment, and other class-related naming information; this information is the responsibility of Behavior's subclass ClassDescription.

To be able to implement its functionality, Behavior has instance variables superclass, methodDict, format, and subclasses. Variable format contains information about the storage layout of the class. methodDict stores associations *selector -> compiled method*. As an example, when you inspect String (a class, and thus subclass of Behavior), you will find the inherited instance variable methodDict among its instance variables (Figure A 4.8) and when you inspect it, you will find that its keys are selectors of all methods defined in String. When you inspect the value of one of these selectors, you will find that it is the compiled method of the corresponding selector - an instance of CompiledMethod.

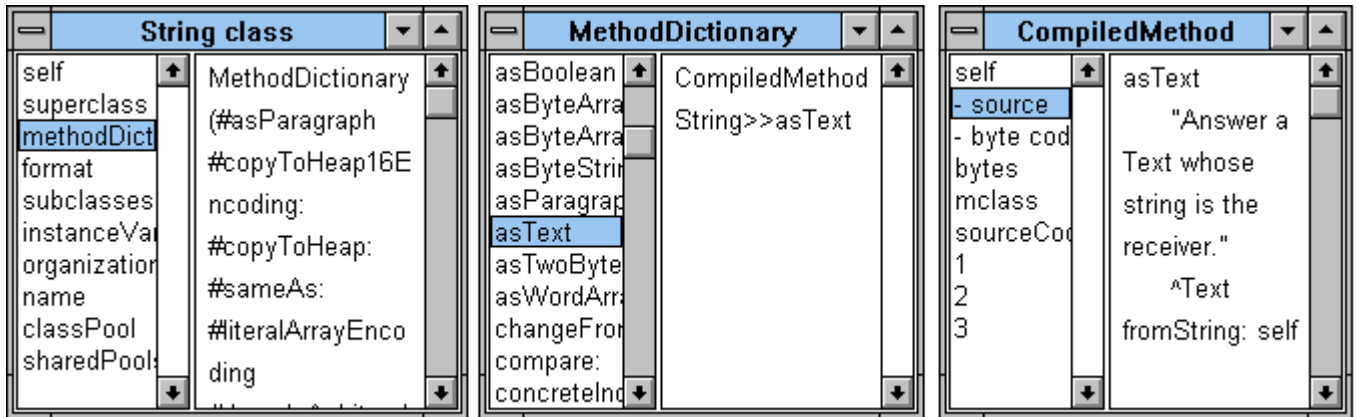


Figure A 4.8. Inspectors revealing instance variable methodDictionary inherited from class Behavior, and its contents.

Class Behavior does not have any class methods and its instance methods are understood by all classes. The following are examples of the most interesting Behavior responsibilities and we encourage you to browse the class for more details.

Instance creation

The most common messages originating in class Behavior are `new` and `new:`. These are the default creation messages inherited (and sometimes redefined) throughout the class hierarchy. Methods `basicNew` and `basicNew:` are their duplicates that should not be redeclared in any class.

Instances, superclasses, subclasses, and enumeration over them

Numerous behaviors are available in this category and we will illustrate a few of them on examples.

To find how many instances of Array currently exist in the environment, execute

```
Array allInstances size
```

To find all subclasses of String execute

String subclasses "Returns #(ByteEncodedString TwoByteString Symbol GapString)."

To find which of all existing arrays has the largest size, evaluate

```
| size |  
size := 0.  
Array allInstancesDo: [:anArray| size := size max: anArray size].  
size
```

or more simply

```
Array allInstances inject: 0 into: [:max :anArray| max max: anArray size]
```

To find all superclasses of a class, execute allSuperclasses as in

```
Set allSuperclasses       "OrderedCollection (Collection Object)."
```

Instance variable format contains a code that describes what kind of class this is. The protocol based on format makes it possible to ask a class whether it has fixed size (classes that have only named variables) or variable size (collections represented internally as indexed elements) and, in the case of a variable size class, whether its variables are stored as eight-bit or 16-bit quantities. As an example, all variable size classes can be obtained by

```
Object withAllSubclasses select: [:aClass| aClass isVariable]
```

which returns

```
OrderedCollection (AnnotatedMethod Array BinaryStorageBytes BOSSBytes BOSSReaderMap ByteArray  
ByteEncodedString ByteString ByteSymbol CCompositeType CCompoundType CEnumerationType  
CEnvironment CompiledBlock CompiledCode CompiledMethod CProcedureType DependentsCollection  
Dictionary Double ExternalDictionary ExternalMethod ExternalRegistry Float FontDescriptionBundle  
GraphicsHandle HandleRegistry HandlerList IdentityDictionary IdentitySet ISO8859L1String LargeArray  
LargeInteger LargeNegativeInteger LargePositiveInteger LargeWordArray LensLinkedDictionary  
LensObjectRegistry LensProtectedLinkedDictionary LensRegistry LensWeakCollection LensWeakRegistry  
LinkedOrderedCollection LinkedWeakAssociationDictionary MacString MarkedMethod MethodDictionary  
MultiValueDictionary ObjectMemory ObjectRegistry OrderedCollection OS2String PoolDictionary  
PropertyListDictionary ScannerTable SegmentedCollection Set SignalCollection SortedCollection  
SortedCollectionWithPolicy SPActiveLines SPSortedLines SystemDictionary TwoByteString  
TwoByteSymbol UninterpretedBytes WeakArray WeakAssociationDictionary WeakDictionary WordArray)
```

As another example of Behavior-defined behavior, statement

```
(Smalltalk select: [:id| id isBehavior and: [id isBits]])
```

returns all elements of Smalltalk that are classes (message isBehavior) and whose instances are stored directly as bits rather than accessed by pointers. It returns

```
SystemDictionary keys (#ByteString #ByteSymbol #GraphicsHandle #ISO8859L1String #Character  
#Double #Float #OS2String #LargeInteger #LargePositiveInteger #TwoByteString #ByteArray #BOSSBytes  
#WordArray #LargeNegativeInteger #UninterpretedBytes #ByteEncodedString #TwoByteSymbol  
#SmallInteger #BinaryStorageBytes #MacString)
```

Expression

```
Object withAllSubclasses select: [:aClass| aClass isBits]
```

would produce the same result. Access to instances of all other classes is by pointers.

Functionality based on access to class hierarchy and method dictionary

Behavior provides several methods for adding new selectors to the method dictionary or removing existing ones, finding method selectors, printing hierarchies, and so on. As an example,

Array printHierarchy

returns a string containing the hierarchy of class Array, the same as provided by the browser. Expression

Point allSelectors

returns a set containing selectors of all messages understood by instances of Point, in other words, names of all instance methods declared in class Point or its superclasses. To find all *class* messages understood by Point, we must ask *metaclass* Point class as in

Point class allSelectors

Behavior also provides access to method source code. As an example,

Collection sourceCodeAt: #contains:

returns the text of the definition of instance method contains:

Similarly, the following expression returns the definition of the *class* method with:

Collection class sourceCodeAt: #with: 'with: anObject

Other protocols

Among other interesting messages, class Behavior provides methods for compiling source code and inserting it into the method dictionary, and a method for decompiling byte codes to produce code with artificially created temporary variables; this method is used by the Debugger when it does not have access to the source code of the selected message. All these methods use classes Compiler and Decompiler to do the actual work. The implementation is very flexible and uses a compiler and decompiler retrieved by an accessing methods so that a different compiler or decompiler may be written and used if desired.

Main lessons learned:

- Behavior is an abstract class that factors out much but not all of the behavior shared by classes and metaclasses.
- Behavior introduces information about a class's superclass and subclasses, its methods, and its format. It does not provide information about instance variables, comment, and protocols.
- The functionality of Behavior is inherited by all classes.

Exercises

1. The all-important `new` and `new:` methods are defined in the instance protocol of `Behavior` and redefined in several other classes. However, `new` and `new:` in `Behavior` are instance methods whereas their use obviously is as a class methods. As an example, we use `Array new: 5`, sending the `new:` message to class `Array`. Moreover, the new definitions in those classes that redefine `new` and `new:` are on the class side of the protocol. This seems to violate the fact that instance methods are inherited on the instance side and class methods on the class side. Explain.
2. Write a code fragment to find all classes that implement `hash` or `=` but not both.
3. Write method `find: aString` to ask the user for a string and find all its occurrences in the source code of all methods declared in the receiver class. As an example, executing `Object find: 'assign'` should return an ordered collection with selectors of all `Object` instance methods whose source code contains the string `'assign'`.
4. Write a code fragment to find all methods in category `Collection-Streams` that disable inherited methods (in other words, all methods in category `Collection-Streams` that use `shouldNotImplement`). Display the names of the methods and their classes in the Transcript.
5. Add a new `<operate>` command called *debug* to open the Debugger and start execution of the selected code. In other words, executing *debug* should work just like adding `self halt` at the beginning of the selection and executing it with *do it*.
6. How many class methods are unary, binary, keyword methods?

A 4.5 Class `ClassDescription`

Class `ClassDescription` is the only immediate subclass of `Behavior`. It provides facilities for naming classes, method protocols, and instance variables, and for class comments. The essence of its comment is as follows:

`ClassDescription` adds a number of facilities to basic `Behavior`:

- named instance variables
- category organization for methods
- most of the mechanism for `fileOut`

`ClassDescription` is an abstract class: its facilities are intended for inheritance by the two subclasses, `Class` and `Metaclass`.

Instance Variables:

`instanceVariables` <Array of: `String`> names of instance fields
`organization` <`ClassOrganizer`> organization of message protocol

As the comment states, `ClassDescription` gathers all behavior shared by classes and metaclasses that is not defined in `Behavior`. Its definition contains many methods:

`ClassDescription allSelectors size`

returns 343, meaning that 343 methods are defined in `ClassDescription`. We leave it to you to discover how to find that there are 12 instance method protocols. The most interesting of them are briefly described below.

Accessing

This protocol provides access to the comment and the standard comment template via messages such as `comment`, `comment:`, and `commentTemplate`.

Copying

This protocol makes it possible to copy a method, a protocol, or all protocols from one place to another. As an example,

Test copyCategory: #accessing from: Collection

copies the accessing protocol from class Collection to class Test. ClassDescription uses the term *category* to refer both to class categories and to method protocols.

Printing

The printing protocol contains methods that return strings with instance variable names, the class definition, and so on. As an example,

Explainer instanceVariablesString

returns the string

'class selector instance context methodText '

Instance variables

This protocol makes it possible to add new instance variables and remove existing ones, and find in which position a named instance variable is stored in the sequence representing a class's instance variables, taking into considerations inherited instance variables. This protocol is needed to understand the byte codes produced by method compilation. As an example,

Window instVarIndexFor: 'sensor'

returns 7 meaning that references to the 7th variable in messages whose receiver is an instance of Window refer to instance variable sensor.

Organization

This protocol contains methods dealing with information about method protocols stored in instance variable organization, an instance of ClassOrganizer. As an example,

Array category

returns #'Collections-Arrayed', the name of the category containing class Array and

View organization categories

returns an array containing the names of the three protocols defined in class View:

#(#'display box accessing' #'controller accessing' #'private')

Since ClassDescription has access to organization, it can also answer questions such as 'Which category (protocol) contains the method with a given selector?' As an example, the following expression returns the name of the protocol containing the class method with:

Collection class whichCategoryIncludesSelector: #with: "Returns #'instance creation'."

File-out

This protocol supports filing out of the source code of the class.

ClassDescription has two subclasses - Class and Metaclass - which define specialized behavior of classes and metaclasses respectively. We will outline their functionality in the next section.

Main lessons learned:

- ClassDescription is a subclass of Behavior whose most visible responsibilities include access to instance variable names, class comment, and organization of instance methods into protocols.
- The combination of Object, Behavior and ClassDescription defines all behaviors shared by classes and metaclasses.

Exercises

1. When you are creating a new test version of a class, you might want to copy all protocols and methods from an existing class to the new class. Write a code fragment to do this.
2. Write a code fragment to find all unreachable methods in a class. (A method is unreachable if it is either never sent or if it is sent by a method that is itself unreachable.)
3. Write a code fragment that lists the names of instance protocols of a selected class in the Transcript. Repeat for class protocols.
4. Modify the default comment that appears in the browser for an uncommented class. The new template should automatically display a list of all instance and class variables following the current default comment.
5. Modify the default comment further to list methods that are subclass responsibility.
6. Write a short summary of the logic of the separation of class and metaclass behaviors into Behavior and ClassDescription.
7. Examples presented in this section show the importance of class ClassOrganizer. Write a short summary of its most important features.
8. The file-out operation is implemented in ClassDescription. Which class is responsible for file-in?

A 4.6 Class Class

Class has the following comment:

Instances of class Class describe the representation and behavior of objects. Class adds more comprehensive programming support facilities to the basic attributes of Behavior and the descriptive facilities of ClassDescription. An example is accessing shared (pool) variables.

Instance Variables:

name	<Symbol> name of class for printing and global reference
classPool	<PoolDictionary nil> of variables common to all instances (i.e., class variables)
sharedPools	<Collection of: Dictionary> access to other shared variables

Two other examples of behaviors that classes need but metaclasses don't are renaming a class (the name of the metaclass is derived from the name of the class) and subclass creation. Class also defines behaviors that classes share with metaclasses but implement differently. These include, for example, adding and removing instance and class variable names, and file out.

Most programmers will never use Class behaviors except for protocol subclass creation which contains methods for defining new classes. Even these methods are rarely used explicitly because new classes are normally defined from the browser template:

```
Paint subclass: #DevicePaint
instanceVariableNames: 'device devicePaint paintBasis '
```

```
classVariableNames: "  
poolDictionaries: "  
category: 'Graphics-Support'
```

Accepting the edited template simply sends message subclass: instanceVariableNames: classVariableNames: poolDictionaries: category: inherited from Class to class Paint and creates subclass DevicePaint. Classes created with this message only have instance variables accessed by word-size pointers. Most classes have this format and respond true to isFixed from class Behavior. The definition of this subclass creation message is worth a look:

```
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat  
"This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver)."  
| approved |  
"Check whether arguments have proper form, such as class variables capitalized."  
approved := SystemUtils  
    validateClassName: t  
    confirm: [:msg :nm | Dialog confirm: msg]  
    warn: [:msg | Dialog warn: msg].  
approved == nil ifTrue: [^nil].  
"Arguments are OK, ask classBuilder to build the class."  
^self classBuilder  
    "First calculate all necessary parameters."  
    superclass: self;  
    environment: self environment;  
    className: approved;  
    instVarString: f;  
    classVarString: d;  
    poolString: (self computeFullPoolString: s);  
    category: cat;  
    beFixed;  
    "Now create new class or revise existing class."  
    reviseSystem
```

The most interesting part is the last line which creates a new class or revises the class if it already exists. Its definition is

reviseSystem

```
"Mutate the system, based on whether the class already exists or not."  
^self needsNewClass  
    ifTrue: [self createNewSubclass]  
    ifFalse: [self modifyExistingClass]
```

If the class is new, message createNewSubclass is executed. Its definition in ClassBuilder is as follows:

createNewSubclass

```
"The class does not exist--create a new class-metaclass pair."  
| newMeta |  
self runValidationChecksForNewClass. "Check that class name is not already in use, etc."  
newMeta := self metaclassClass new. "Create new instance of Metaclass and initialize it."  
newMeta assignSuperclass: (self classOf: superclass).  
newMeta methodDictionary: MethodDictionary new.  
newMeta setInstanceFormat: (self classOf: superclass) format.  
class := newMeta new. "Ask the metaclass to create the class."  
class assignSuperclass: superclass. "Initialize instance variables of the class."  
class methodDictionary: MethodDictionary new.  
class setInstanceFormat: self computeFormat.  
self setStructureOf: class.  
class setName: className.  
self register: class inPlaceOf: nil.
```



```
self changeMicroState. "Aspects such as category and pools."  
^self logNew: class    "Record new class so that changes will be saved on exit."
```

To define a class that does not fall into the usual pattern of fixed-size class with named instance variables, edit the Browser template into one of the following forms

variableByteSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:

variableSubclass instanceVariableNames: classVariableNames: poolDictionaries: category:

edit it in the usual way, and *accept*.

If the class category given as the argument does not exist, the message creates it.

Example: Finding which symbols in Smalltalk do not represent classes

Problem: The system dictionary Smalltalk includes all global objects. They are mostly classes but also objects of other kinds. This example explores what these other objects are.

Solution: Since all class objects are subclasses of Class, we can use the following expression to extract the non-Class objects from the Smalltalk dictionary:

```
Smalltalk reject: [:value| value isKindOf: Class]
```

This returns the following ten or so elements out of the 1200 or so associations in Smalltalk

```
SystemDictionary keys (#SymbolicPaintConstants #Processor #X11InputManagerDictionary  
#TextConstants #Transcript #IOConstants #OpcodePool #Smalltalk #InputManagerDictionary #Undeclared  
#ScheduledControllers #NullInputManagerDictionary )
```

and includes some familiar global variables such as Transcript, Processor, and ScheduledControllers, and some pool dictionaries such as TextConstants and Undeclared. Undeclared contains all variables that the user did not declare when compiling a program and specified that they should be left undeclared when responding to the compiler query. We leave it to you to find what all these objects are.

Main lessons learned:

- Class inherits the behaviors of Behavior and ClassDescription and adds knowledge of class name and access to class variables and shared pools. It also provides facilities for creating new classes and controlling their format (fixed size, variable size, and others).

Exercises

1. Why is the name of a class defined in Class rather than Behavior or ClassDescription?
2. Trace the creation of a new class and write a short summary.
3. What happens when you redefine and accept an existing class?
4. What is the use of the global pool dictionary Undeclared?

A 4.7 Class Metaclass

Class Metaclass is the equivalent of Class for metaclasses. It has the following comment:

Metaclasses add instance-specific behavior to various classes in the system. This typically includes messages for initializing class variables and instance creation messages particular to that class. Metaclass has only one instance. A metaclass shares the class variables of its instance.

Instance Variables:

thisClass <Class> the chief instance of the receiver, which the receiver describes

Metaclass mainly redefines behaviors which are different from Class behaviors. One example is `definitionMessage` which is responsible for providing the text displayed in the Browser for the definition of a class in the instance view, or its metaclass in the class view. Another example is the `new` message which is automatically invoked to create a metaclass during the creation of a new class. Yet another example is the enumeration message `allSubclassesDo:` which must enumerate over metaclasses, unlike its class version defined in `Behavior`.

Main lessons learned:

- Just as `Class` defines and inherits or redefines behavior needed by classes, `Metaclass` defines, inherits, and redefines all behavior common to metaclasses.

Exercises

1. As we know, class objects are stored in the Smalltalk SystemDictionary. Where are metaclasses stored?
2. We have now covered the principles of the whole class hierarchy except for its essential class - `Object`. What is the superclass of `Object` and what implications does it have?

A 4.7 Metaprogramming - Is this magic useful?

After the first section of this chapter, you may have been wondering whether the subtle architecture of classes provides anything beyond a proof that an environment consisting only of objects can indeed be constructed. By now, it should be obvious that metaclasses and the whole structure of the class hierarchy have important practical roles. First, classes such as `Behavior` and `ClassDescription` are heavily involved in the creation of new classes and modification of existing ones. Second, these classes provide access to very valuable information about classes and their instances, and environment tools such as the Browser very much depend on them. Third, programming based on kernel classes and other classes such as compiler classes can be used to extend the programming environment (for example, by adding new browsers), the environment of a running application (for example, by adding new methods or classes at run time), or even the syntax or semantics of the Smalltalk language. This form of programming is called *metaprogramming*¹ and we will now demonstrate it on several examples.

Example 1: A creation method to create a class and its accessing methods²

The Browser's class definition template makes it possible to create a new class with variables. Most instance variables have accessing methods and these methods must be created manually. Since Smalltalk provides access to its compiler and all class information, it is possible to create tools to create accessing methods automatically.

Problem: Define a new class creation message that creates a class and all accessing methods for its instance variables. The usage of the method should be as in

¹ Languages that allow metaprogramming are called *reflective*. Most languages are not reflective.

² This example is based on the classic 'blue book' by Goldberg and Robson, a Smalltalk bible that should be read by anybody seriously interested in Smalltalk.

```
Object subclassWithAccessors: NewClass
  instanceVariableNames: 'x y z'
  classVariableNames: 'X Y Z'
  poolDictionaries: 'Pool1 Pool2'
  category: 'Experimental'
```

in other words, the same as the standard Browser template

```
Object subclass: NewClass
  instanceVariableNames: 'x y z'
  classVariableNames: 'X Y Z'
  poolDictionaries: 'Pool1 Pool2'
  category: 'Experimental'
```

except that the first keyword is different.

Solution: The algorithm is simple:

1. Create the class using the existing creation message.
2. Generate the source code of accessing methods and ask the class to format it.
3. Compile the methods. This will automatically insert them into the method dictionary.

The implementation of this algorithm (a method to be added to class Class) is relatively simple:

```
subclassWithAccessors: t instanceVariableNames: f classVariableNames: d poolDictionaries: s  
category: cat
```

```
"Create new class and its accessing methods."
```

```
| newClass |
```

```
"Compile class with its variables using existing creation message."
```

```
newClass := self subclass: t
  instanceVariableNames: f
  classVariableNames: d
  poolDictionaries: s
  category: cat.
```

```
"Formulate, format, and compile accessing methods for all instance variables."
```

```
newClass instVarNames
```

```
do: [:aName | "Enumeration over all instance variables."
```

```
| formattedText |
```

```
"Construct the text, format it, and compile it into the accessing protocol."
```

```
"First the 'get' method."
```

```
formattedText := Compiler new
```

```
  format: aName , '^ ' , aName
```

```
  in: newClass
```

```
  notifying: nil.
```

```
"Now the 'set' method."
```

```
newClass compile: formattedText classified: #accessing.
```

```
formattedText := Compiler new
```

```
  format: aName , ': argument ' , aName , ' := argument'
```

```
  in: newClass
```

```
  notifying: nil.
```

```
newClass compile: formattedText classified: #accessing]
```

Main lessons learned:

- A programming environment is called reflective if it provides information about its implementation and allows the programmer modify it.
- Programming based on reflectivity is called metaprogramming.

Exercises

1. Test that the class creation method developed in this section works.
2. Add a new command called *accessing* to the <operate> menu of the text view of the Browser. Its activation should open a window containing two parallel multiple selection lists containing all instance variables of the currently selected class and a *Compile* button. The left list is for creating 'get' methods, the right list is for creating 'set' methods. Selecting variable names in the two lists and clicking *Compile* creates accessing methods for all selections.

A 4.8 Enhanced Workspace - Another example of metaprogramming

The functionality of the Workspace would be greatly enhanced if it could hold on to objects created by program fragments, in other words, if results of evaluating selected messages could persist for the duration of the existence of the workspace. Such objects could then be reused and we would not have to recreate them every time. We will show how this idea could be implemented, leaving a more complete implementation as a project.

Problem: Implement an enhanced workspace with an additional <operate> menu command *do and keep* that will execute just like *do it* but save the resulting object as a persistent part of the workspace and bind it to an automatically generated variable. An additional pop up command called *variables* will open a multiple choice menu on all variables created in this way, and allow the user to select and inspect one. Within the scope of the workspace, these variables will be treated as global.

Solution: We will create the Extended Workspace as a new application, an instance of a new class called *ExtendedWorkspace*, a subclass of *ApplicationModel*. In its design, we will consider two basic scenarios: executing a code fragment to create a workspace-persistent object, and executing a code fragment containing a reference to such an object via the variable name associated with it.

Scenario 1: Creating a workspace-persistent object

Since there is no limit on the number of objects that the user may want to add to the extended workspace, the obvious approach is to store them in a dictionary whose key is the name of the variable and whose value is the object. However, this implementation would force us to refer to the object as a value in a dictionary as in

workspaceVars at: varname

rather than as a variable as in

x := y factorial

To be able to access the persistent objects as named variables, we can use one of at least the following two approaches: Create the new variable as a global variable (adding it to Smalltalk), or add it as a named instance variable to the *ExtendedWorkspace* at run time.

Using the first approach, we could automatically generate numbered variables, such as *Var0001*, *Var0002*, and so on, and delete them when the user closes the workspace. This approach is relatively simple but it does not restrict the variable to one workspace; we leave it as an exercise.

The other approach is to add a new named instance variable to the *ExtendedWorkspace* class every time when the user requests a new persistent object; this is the strategy that we will use here. This approach requires that we modify the definition of the extended workspace class at run time every time when the user adds a new workspace-persistent object. Since there may be any number of workspaces, each of them with a different number of variables, we will do this by treating *ExtendedWorkspace* as an abstract class, creating a new numbered subclass called, for example, *ConcreteExtendedWorkspace7* when we open a new workspace, and adding a new instance variable to this class whenever we add a new persistent object to this workspace. This way, different workspaces will be instances of different classes and their

'persistency' variables will be mutually unknown. The new class will be automatically deleted when the user closes the workspace.

This principle can be implemented as follows: When the user creates a new workspace-persistent object, 'mutate' the corresponding `ConcreteExtendedWorkspace` class by adding a new instance variable to it, and bind the object to this variable. This procedure is relatively simple as it only requires one message from `Class` to perform the mutation, one message to add an accessing method for the new variable, and one message to access the variable and assign the new value to it.

To test the idea of mutation, assume the existence of a test class called `MutationTest`, a subclass of `Object`, with one instance variable called `var1` and its accessing message `var1:`. Our program will assign a value to `var1` using this accessing message, add new variable `var2` and its accessing method, assign a value to this variable, and inspect the result:

```
| x |
"Create an instance of the test class and assign a value to var1."
x := MutationTest new.
x var1: 10.
"Mutate the class by adding a new variable."
MutationTest addInstVarName: 'var2'.
"Add its accessing method."
MutationTest compile: 'var2: anObject ^var2 := anObject'
               classified: 'accessing'
               notifying: nil.
"Assign value to the new instance variable."
x perform: #var2: with: 20.
"x var2: 20 would not work, var: does not exist as this code fragment is being compiled."
x inspect
```

Everything works as we hoped: The `x` object has both `var1` and `var2` variables and their values are as expected. You can also open the browser on `MutationTest` to see that the new definition of `MutationTest` captures the changes made by this program.

The principle thus works and we can apply it to our problem. In our implementation of `ExtendedWorkspace`, the value assigned to the new variable will be the object that we want to make persistent and the mutation will be performed when the user executes, for example,

```
| x y |
x := 130 * 60 cos.
y := 130 * 60 sin.
^x @ y
```

with *do and keep*. This will create a new workspace-defined persistent variable and store the `Point` in it.

We now have the solution to the problem of creating workspace-persistent objects and their associated variables. Our next problem is how will these variables be accessed when we execute a program in the workspace.

Scenario 2: Executing a code fragment containing a reference to a persistent object

Assume that the user created two persistent objects and assigned them to automatically created variables `var1` and `var2`. Assume that the user now executes the following fragment in the extended workspace using, for example, *do it*.

```
| radius |
radius := (var1 squared) + (var2 squared)
...
```

To make this work as we wish, we must first understand how *do it* works. When we examine implementors of `dolt`, we find the following definition in class `ParagraphEditor`:

dolt

```
"Evaluate the current text selection as an expression."  
self selectionStartIndex = self selectionStopIndex  
ifTrue: [^self]. "If the current selection is empty, just return."  
self class compilationErrorSignal  
  handle: [:ex | ex returnWith: nil]  
  do: [self evaluateSelection]
```

The evaluation of the selection is obviously done by `evaluateSelection` whose definition

evaluateSelection

```
"Evaluate the current text selection as an expression"  
| result |  
result := Cursor execute showWhile:  
  [self doltReceiver class evaluatorClass new  
    evaluate: self selectionAsStream  
    in: self doltContext  
    receiver: self doltReceiver  
    notifying: self  
    ifFail: [self class compilationErrorSignal raise]].  
self doltValue: result.  
SourceFileManager default logChange: self selection string.  
^result
```

shows that evaluation uses the stream which is the first argument of `evaluate:in:receiver:notifying:ifFail:`. When you examine its receiver, you will find that `evaluatorClass` returns `SmalltalkCompiler` and the receiver thus amounts to `SmalltalkCompiler new`. Expression `self selectionAsStream` appears to refer to the highlighted text that is being evaluated. However, this text only includes the variables defined in the code fragment. Can we include our workspace-based variables as the context of evaluation of the code fragment? The definition of `evaluate:in:receiver:notifying:ifFail:` is as follows:

evaluate: textOrStream in: aContext receiver: receiver notifying: aRequestor ifFail: failBlock

"Compiles the sourceStream into a parse tree, then generates code into a method. If receiver is not nil, then the text can refer to instance variables of that receiver ... etc.

and this shows that if we use our extended workspace as receiver, the evaluated code can make references to our workspace-based variables.

The next question is how to access the `dolt` method, in other words, how to get a suitable instance of a `ParagraphEditor`. Using the Browser, we find that `ParagraphEditor` is the controller (the object that handles user input) of its text editor. To obtain the controller, we must ask the text editor widget as in

```
(aBuilder componentAt: #textEditor) widget controller
```

However, if we just sent `dolt` to the `ParagraphEditor`, `dolt` would use its definition of `doltReceiver` which returns nil and this is not what we want. This means that we must define a new controller with a new definition of `doltReceiver` that returns the application model of our workspace which contains the our workspace-based variables.

We will thus define a subclass of `ParagraphEditor` containing only a new definition of `doltReceiver` that returns the application model. We will then assign this new controller to our text widget instead of the default `ParagraphEditor`. This opens two questions: How do we assign a new controller, and how do we get from the controller back to the application model.

Assigning a new controller to a widget is simple - just send `controller:` to the widget as in

```
(aBuilder componentAt: #textEditor) widget controller
```

To see how to get from a controller back to the application model, we will create a test application whose window contains only a text editor. In its `postBuildWith:` method, we will open an inspector on the builder:

postBuildWith: aBuilder
aBuilder inspect

In the inspector, we can locate the controller of the widget and trace how to get back to the application model. We find that the controller has a reference to its view which is wrapped in several wrappers, the last of which is a part of a window and this window can access the builder. We can thus access the builder and its model, and this is the object that should be returned by `doltReceiver`. We leave it to you to implement the whole idea.

Exercises

1. Implement `ExtendedWorkspace`. Don't forget to delete the class when the user closes the workspace.
2. Reimplement `ExtendedWorkspace` using global variables. Don't forget to delete the global variables when the user closes the workspace.
3. Modify `ExtendedWorkspace` so that the user can delete selected workspace-persistent variables.
4. Modify `ExtendedWorkspace` so that the user can rename workspace-persistent variables.
5. Add an inspector view for workspace variables at the bottom of the extended workspace.

A 4.9 Another example: Wrapping objects to intercept messages

A common debugging problem is that an object is changing in some undesirable way and you don't know where. You can, of course, put a breakpoint in front of the creation message and trace your program message by message until you find where the problem occurs but this may be very time-consuming. A much better solution is to intercept all messages sent to the object in question and to open the debugger or perform some other action when such a message occurs.

If we have complete control over the object, the solution is simple - just insert `self halt` in all methods that refer to it. Very often, however, the object is a system object or there may be too many references to it. In such a case, we would like to have a tool that allows us to create this interception mechanism automatically.

One way to intercept messages to object *x* is to insert another object 'in front' of it - a 'proxy' as in Figure A 4.9. All messages to *x* then go to this proxy which can process them in an arbitrary way, for example opening the Debugger and allowing the user to pass the messages on to *x*.

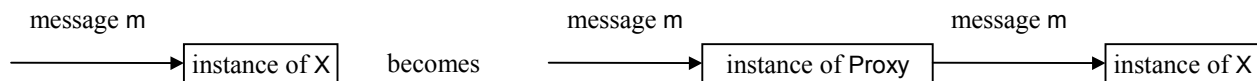


Figure A 4.9. Intercepting messages to *x* by a proxy.

A possible solution is as follows: Assume that the object in question (call it *x*) is an instance of class *X*. Modify creation messages of class *X* so that they create an instance of *X* but return an instance *p* of *Proxy* that knows about *x*. All communication with the returned object will now go to *p*, and *p* can direct it to *x* and perform any other operations specified by the programmer.

Having formulated the principle, let's consider how we can implement it. The obvious way is to define *Proxy* so that it understands all messages understood by *x* (defined in *X* and all its superclasses). These messages would have to be modified to allow the programmer to execute the programmer-specified action and then pass the message on to *x*. Although this approach can be automated, it is very unattractive because it requires defining possibly hundreds of messages in *Proxy*.

An interesting alternative to making sure that *p* understands all messages of *x* is to make sure that *p* understands as few messages of *x* as possible. When a message understood by *x* is then sent to *p*, *p* does not understand it, this sends `doesNotUnderstand: aMessage` to *p*, and if we redefine `doesNotUnderstand:` in

Proxy to insert a programmer action and pass the original message to x, the problem is solved (Figure A 4.10). This approach is very simple and we will now implement it.

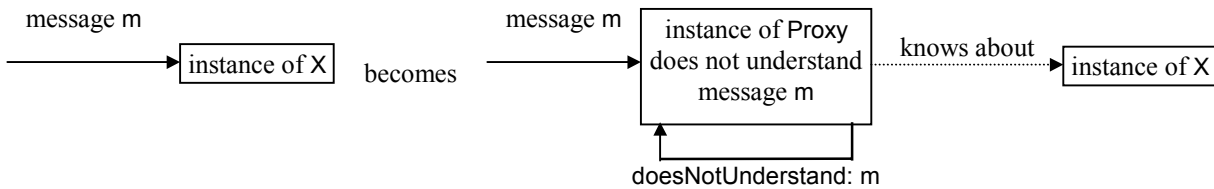


Figure A 4.10. Intercepting messages to x by a proxy via doesNotUnderstand:.

The essential question in defining Proxy is what should be its superclass. Since we want it to share as few methods with class X as possible, we must place it as high in the hierarchy as possible so that it inherits as few messages as possible. This seems to suggest that Proxy should be a subclass of Object. But if it is, it will still understand all Object's messages and these quite likely include messages that we would like to intercept. As an example, if X is an Array, we may want to intercept `at:put:` - but this message is defined in Object. We thus want to eliminate even inheritance from Object. But if we don't want even Object to be the superclass of Proxy, then what should the superclass be? The obvious question is *nothing* - the proxy should not have a superclass - just like Object which also does not have a superclass. In other words, in place of a superclass, use nil. And this is exactly what we will do.

Having decided this, the next question is what behaviors should Proxy implement and what attributes it needs. Instances of Proxy must obviously have access to the instance of the original addressee of X messages; we will call its instance variable `addressee`. The only behavior that appears necessary is a new definition of `doesNotUnderstand:` that allows the programmer to specify a block to be executed before the intercepted message is passed to addressee. To understand the possible implications, let's examine the existing definition of `doesNotUnderstand:` in class Object:

doesNotUnderstand: aMessage

"The default behavior is to create a Notifier containing the appropriate message and to allow the user to open a Debugger. Subclasses can override this message in order to modify this behavior."

```
| selectorString |
    selectorString :=
        Object errorSignal
            handle: [:ex | ex returnWith: '** unprintable selector **']
            do: [aMessage selector printString].
    Object messageNotUnderstoodSignal
        raiseRequestWith: aMessage
        errorString: 'Message not understood: ', selectorString.
    ^self perform: aMessage selector withArguments: aMessage arguments
```

The argument is an instance of class Message with information about the selector of the intercepted message and its argument. `doesNotUnderstand:` first attempts to convert the selector to a string and raises an exception (Chapter 11) if it fails. It then opens the usual exception window, and if the user selects *proceed*, it attempts to execute the original message. This normally fails but in our case, we can take advantage of this behavior to pass the message to addressee or execute another action.

We must now decide what should happen when p intercepts a message to x. We may or may not want to open the exception window (we might, for example, only want to collect some statistics about messages to x without ever opening the Debugger). To make this possible, we will create p with a Boolean specifying whether to open the exception window or not.

The next thing we want is to be able to execute an arbitrary block because we don't know what the user might want to do with the intercepted message. What should this block be allowed to do? We cannot know, but in principle it should be able to do anything with the information now available which includes: `addressee`, `selector`, and `arguments`. We will thus assume that the block is a three-argument block with these three objects as its arguments, to provide the maximum possible flexibility.

The design is now complete and we are ready to implement Proxy. Its definition is


```
nil subclass: #Proxy
  instanceVariableNames: 'addressee doesNotUnderstandBlock openNotifier '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'tests - proxy'
```

Instance variable `doesNotUnderstandBlock` is the three-argument block to be executed when a message is intercepted, and `openNotifier` is a Boolean that determines whether the intercepted message should open an exception window or not. *Accepting* this definition in the Browser will open a confirmation window because `nil` should not normally be the superclass; click *OK* and proceed.

The creation message for `Proxy` must create a new `Proxy` object with an addressee, a block, and information about whether to open an exception window or not:

newOn: anObject block: aBlock openNotifier: aBoolean

```
^(self new) addressee: anObject; block: aBlock; openNotifier: aBoolean
```

Finally the `doesNotUnderstand:` method. In `VisualWorks`, compilation of a subclass of `nil` automatically adds a few essential methods and these include the standard `doesNotUnderstand:`. We change it according to our earlier analysis as follows:

doesNotUnderstand: aMessage

"Provide the default behavior but add control over the exception window, execution of a programmer supplied block, and finally execution by addressee."

```
(Object canUnderstand: aMessage selector)
  ifTrue:
    [self class copy: aMessage selector from: Object.
     ^self perform: aMessage selector withArguments: aMessage arguments].
openNotifier ifTrue: [Object messageNotUnderstoodSignal raiseRequestWith: aMessage
  errorString: 'Message not understood: ', aMessage selector].
doesNotUnderstandBlock
  value: aMessage
  value: aMessage arguments
  value: addressee.
^addressee perform: aMessage selector withArguments: aMessage arguments
```

This completes the definition of `Proxy` and we are now ready to test it. To do this, we will define a new class called `TestProxy` with a single instance variable called `var` and a single instance method

var: anObject

```
var := anObject
```

Assume that we don't want to open an exception window with each message to a `TestOfProxy`, and that we want it to print which object is being addressed, what is the selector of the intercepted message, and what are its arguments. For the purpose of this test, we will thus write a new `TestOfProxy` creation method defined as follows:

newWithProxy

"Create a proxy that does not open exception window and prints information about myself and the intercepted message."

```
| newInstance |
newInstance := self new.
^Proxy newOn: newInstance
  block: [:message :arguments :addressee | Transcript cr; show: addressee printString ,
    'gets message ' , message selector , ' with arguments: ' , arguments printString]
  openNotifier: false
```

To test our scheme, execute

TestOfProxy newWithProxy var: 13; var: 15

and you will get the expected result. When you are finished testing, remove this method so that TestProxy again behaves 'normally'.

Main lessons learned:

- It is possible to define classes outside the Object hierarchy. In fact, one can create another completely separate hierarchy by using nil as the 'superclass' of the root of this hierarchy.

Exercises

1. Write a program to run two series of tests of Array methods using Proxy. In the first series, print a log of message sends by your program in the Transcript. In the second series, let each intercepted message open an exception window.
2. The idea of parallel class hierarchies rooted in nil is used in the complete VisualWorks library. Find all classes in the library whose superclass is nil.
3. Use Proxy to display a confirmer with the name of the intercepted message (Figure A 4.11). An exception window opens only if the user clicks *yes*.
4. Proxy will be even more powerful if the programmer can specify an additional object to be used in the block. This object could, for example, collect some statistics. Extend the definition of Proxy accordingly.
5. Automate the proxy creation mechanism as much as possible.
6. Write a method that will insert a breakpoint into every method in a given class that contains a reference to a given instance variable. Write also a method that will remove all these breakpoints.



Figure A 4.11. Confirmer for Exercise 3.

Conclusion

Every class in Smalltalk is the single instance of a matching metaclass which is automatically created when the class is first compiled. Metaclasses use the same names as their corresponding classes and cannot be accessed through the System Browser in the same way as 'regular' classes. Instead, the `class` message must be addressed to the corresponding class.

Metaclasses form a hierarchy that parallels the hierarchy of classes, with the metaclass of `Object` at the top. Metaclass `Object`, however, has itself several superclasses that define behavior shared by all classes and all metaclasses, and their top superclass is class `Object`. Class `Object` thus remains a superclass of all classes, including metaclasses.

Like all classes in Smalltalk, each metaclass is itself an instance of a class. Each metaclass is an instance of class `Metaclass`.

Whereas class `Metaclass` defines behaviors particular to metaclasses, behaviors particular to classes are defined in class `Class`. Much of the behavior of `Metaclass` and `Class` is shared and defined in class `ClassDescription` which is a subclass of `Behavior`, itself a subclass of `Object`. As the names suggest, `Behavior` defines functionality related to basic class behaviors such as methods for creating new classes. `ClassDescription` adds information about details of class descriptions such as comments and variables. `Class`, `Metaclass`, `Behavior`, and `ClassDescription` are grouped together in category `Kernel Classes`.

Although the basic class hierarchy has `Object` as its root, it is possible to create additional parallel hierarchies rooted in new classes whose superclass is undefined - `nil`.

On the theoretical and esthetic side, the hierarchy of metaclasses and kernel classes gives a constructive proof that a system can be designed consisting of objects and nothing else. On the practical side, these classes provide many essential behaviors that make possible Smalltalk programming tools such as the Browser, allow creation of classes and access to their descriptions, and provide a very powerful mechanism that makes it possible to reshape the Smalltalk environment from within, even at run-time. This property is called *reflectivity* and programming based on reflectivity is called *metaprogramming*. Reflectivity and metaprogramming make Smalltalk one of the most powerful programming environments.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Behavior, *Class*, *ClassDescription*, *ClassOrganizer*, *Metaclass*.

Terms introduced in this chapter

metaclass - the class of a class; accessed by sending *class* to class name

kernel class - a class providing access to shared class behavior and system organization

metaprogramming - programming based on kernel classes

reflectivity - features of a programming language allowing the programmer to modify the language and its programming environment programmatically

Appendix 5 - Style recommendations

Overview

This appendix contains a small selection of Smalltalk conventions, idioms, and patterns. The stylistic part contains a few simple and natural guidelines for naming variables and methods, writing comments, naming method protocols, and similar topics. The idioms and patterns part gives examples of some of the techniques used by experienced Smalltalk programmers.

In some details, stylistic guidelines and preferred patterns vary from one programmer to another and some of our suggestions thus differ from those recommended by some other authors. In the end, it is not important whether you follow our recommendations or somebody else's because what matters is to have a stable style rather than follow one particular style.

A 5.1 Introduction to Smalltalk style guidelines

Having a set of rules for writing Smalltalk programs makes programs easier to read and understand and this is very important for readability and understandability. If you are a part of a team or a student in a course, setting a common set of stylistic guidelines is essential. Most of the following guidelines have been extracted from a much larger set listed and justified in [Skublics, et al.] and [Beck]. We recommend these two highly respected books as additional reading.

A 5.2 Naming

Guideline N.1: Uppercase and lowercase first letter.

Start names of shared objects with upper case letter (classes, global variables, class variables, pools), start names of methods, keywords, parameters, and local names (instance, temporary, and block variables) with lower case.

Examples: Shared objects - Number, Smalltalk, Pi, TextConstants; methods and local objects - factorial, between:and:, width.

Guideline N.2: Multiword names.

When a name combines several words, start each word with a capital letter.

Examples: SmallInteger, studentNumber.

Guideline N.3: Choice of name.

Use short descriptive names and avoid abbreviations except where standard.

Examples: Number, today, factorial, between:and:, BTree (balanced tree)

Guideline N.4: Names of methods.

Method names should suggest the kind of object being returned.

Examples: isHungry, name, between:and:

Guideline N.5: Names of keywords.

Use names describing the role of parameters.

Examples: between:and:, name:address:

Guideline N.6: Variable names versus argument names.

Variable names should indicate the role of the variable, argument names should indicate the kind of argument expected.

Examples: Variable names - width, seed; argument names - name: aString, + aNumber

Guideline N.7: Argument names when type is repeated.

When several arguments are of the same type, distinguish them by indicating their role.

Example: origin: originPoint corner: cornerPoint

Guideline N.8: Names of block arguments.

Name block arguments according to their role.

Example: employees do: [:employee| ...]

Guideline N.9: Names of accessing methods.

Name accessing methods by names of accessed variables.

Examples: Use width, width: to get or set variable width.

Guideline N.10: Names of state-related test methods.

Use a state-related phrase for tests and access of Boolean variables.

Examples: isEmpty, hasBorder, isOn

Guideline N.11: Names of action methods.

For methods that perform an action, use imperative form of verbs describing effect.

Examples: display:, enable, turnOn, turnOff

Guideline N.12: Names of widget properties.

Use widget label to name *Action*, *Selection*, and *Aspect* properties.

Examples: Use action method **save** for action button named *Save*, use selection property **red** for check box named *Red*, use aspect named **name** for input field with label *Name*.

Exercise

1. Browse the class library and find ten well selected names and five poorly selected names.

A 5.3 Comments

Guideline C.1: Use meaningful, short, and grammatically correct comments using correct spelling, full sentences, and standard English punctuation. Use active voice.

Example: "Create and return a Point at the given radius and angle."

Guideline C.2: Follow the Browser method template: Start all methods except accessing and very short self-explanatory methods with a comment explaining some or all of the following: the purpose of the method, its principle, its prerequisites, its effect, its use, the role of its arguments, the returned object.

Example: "Include all elements of aCollection as the receiver's elements. Answer aCollection."

Guideline C.3: Minimize the number of comments in method body - code should be self-explanatory.

Guideline C.4: For class comment, use the template displayed in the browser for class comment. Use active voice. If you wish to use first person, use it consistently.

Example: See comment of class Array, Float, Date for examples.

Exercise

1. Browse the class library and find five good and bad method and class comments.

A 5.4 Formatting

Guideline F.1: Use automatic formatting. Since the VisualWorks formatter exhibits some undesirable behavior, such as misplacing comments, we recommend using the formatter written by Randy Giffen and included in our software package. If your formatter allows several formatting styles, use one consistently.

Exercise

1. Write a method without any regard for formatting and format it. Compare the readability of the two versions.

A 5.5 Names of common protocols

Wherever appropriate, use the following protocol names common in the class library:

Class protocols

class initialization
examples
instance creation
interface spec
resources

Instance protocols

accessing
adding
arithmetic
converting
comparing
controller accessing
copying
events
displaying
enumerating
menu commands
printing
private
private - menu messages
removing
testing
updating

Exercises

1. Browse the class library and find the proportion of the above names among protocol names.
2. Browse the class library and find ten non-standard protocol names.

A 5.6 Introduction to idioms and patterns

Over the more than 20 years of Smalltalk existence, Smalltalk programmers invented and tested various ideas and evolved a body of rules that help writing readable, easily extendible, and efficient code. Why not take advantage of this body of knowledge and save yourself months of frustrating trial and error?

Most of the following guidelines have been borrowed from [Beck] and [Skublics, et al.]. Our list is limited to idioms and patterns used in this book and we recommend highly that you read the recommended references for more details and many more guidelines.

Some of the following guidelines can be classified as patterns, meaning that their validity extends beyond Smalltalk. Other guidelines can be classified as idioms because they are specific to Smalltalk and don't apply to other languages.

A 5.7 General patterns

Pattern G.1: Don't Say Anything Twice

When the same piece of code appears in several places in one class, convert it to a method. When the same object is calculated more than once in one method, cache it in an instance variable. When the same object is needed in several methods in the same class, create an instance variable to hold it.

Pattern G.2: Lots of Little Pieces

Shorter methods, and objects with fewer components are better. They are easier to understand and more reusable. Like all smaller things, they don't 'break' so easily to become useless.

Pattern G.3: Think of Alternatives

When designing a system or a class think of extensions and modifications. Can they be done by adding a few new classes or will some classes have to be rewritten?

Pattern G.4: Think of Specialization

If you anticipate subclassing, separate code into methods that will be inherited in their entirety from methods that will be redefined in subclasses.

A 5.8 Methods

Idiom M.1: Short Method

Keep methods to a few lines. Most methods should fit into the Browser window.

Idiom M.2: Self-Contained Method

A method should perform one identifiable task.

Idiom M.3: Composed Method

When a method gets too long, divide it into self-contained methods, all at the same level of abstraction.

Idiom M.4: Constructor Method

Creation methods should create ready-to-use objects. When appropriate, provide methods that create instances with initialized instance variables.

Example: Rectangle class >> origin: originPoint corner: cornerPoint rather than just Rectangle new.

Idiom M.5: Debugging Method

To support debugging, write a printOn: method early in the coding stage of a new class.

Idiom M.6: Display Method

To support display of selected aspects of an object in a widget such as a list, write a displayString method.

Idiom M.7: Selection Avoiding Method - Polymorphism

To avoid complicating code with multiple ifTrue:ifFalse: messages, use polymorphism - define the same selector with different implementations in all classes whose instances may be accessed in the same way.

Example: Geometric objects define methods displayStrokedOn: and displayFilledOn: and may thus be used interchangeably in code including enumeration. This avoids the need for testing what kind of geometric objects is being displayed, speeding up operation, making code more readable and more easily extendible.

Idiom M.8: Double Dispatch Method

To avoid complicating code with multiple `ifTrue:ifFalse:` messages when the combination of the receiver and the argument determine the operation, send a message to the argument.

Example: Implementation of `+` `aNumber` uses `aNumber sumFromInteger: self` in `SmallInteger`, `aNumber sumFromFloat: self` in class `Float`, and `aNumber sumFromFraction: self` in class `Fraction`

Idiom M.9: Simple Delegation

When an object needs work involving one of its components, delegate the message to the component.

Example: The `printOn:` method in `Point` sends `printOn:` to the `x` and `y` components. Note: A stricter definition of delegation classifies this principle as forwarding.

Idiom M.10: Self Delegation

When the delegate needs to know about the delegating object, send message with a `for:` keyword and `self` as the argument.

Example: A dialog window may need access to the instance variables of the application model that created it. If the dialog's model is different from the application model, open the modal dialog with `openFor: self` from the main application, thus providing access to the main application's state.

A 5.9 Behaviors

Idiom B.1: Instance Variable Enumerator

To enumerate over selected instance variables, create accessing methods for each variable, and use an array of symbols representing the variables' accessing methods and enumerate using `perform:`.

Example: Assume that we have variables and accessing methods called `part1`, `part2`, `part3`, `part4`, `part5`. To calculate the total of part values, use

```
^(#part1 #part2 #part3 #part4 #part5) inject: 0 into: [:part :total| (total + self perform: part) value]
```

Idiom B.2: Pluggable Behavior

When the method to be executed is not known beforehand, for example when the desired method depends on context, use one of the forms of `perform:` with a symbol representing the method.

Examples: Operation of widgets is based on using their aspect accessing method via `perform:` because the method needed to execute the method is not known until the user defines it as the widget's property. Similarly, execution of pop up menu commands is based on `perform:`.

Idiom V.3: Redefining Equality

If a new class cannot reuse the inherited equality, redefine `=` and `hash`. Both should use equality and `hash` of those instance variables that are relevant for comparison. The `hash` method frequently uses `xor:` or `+` on the components used for `=`. Remember that *two objects that are equal must have the same hash value* (but not necessarily the other way around).

Examples: Equality in class `ColorValue` is defined as

= aColor

```
^aColor class == self class and:  
  [aColor scaledRed = red and:  
    [aColor scaledGreen = green and:  
      [aColor scaledBlue = blue]]]
```

and because it uses colors, `hash` is redefined as

hash

```
^red hash + green hash + blue hash
```

Class `Rectangle` redefines `=` in terms of equality of corners and origins, and `hash` is thus defined as

hash

```
^origin hash bitXor: corner hash
```

Idiom B.4: Inclusion Test Reversal

Testing whether an object has one of several possible values can often be performed more easily by testing whether the collection of all possible values includes the object.

Examples: The first impulse for testing whether a `Character` is a vowel is

isVowel

| lowercase |

lowercase := self asLowercase.

^lowercase = \$a | lowercase = \$e | lowercase = \$i | lowercase = \$o | lowercase = \$u

but a better solution is

isVowel

^#(\$a \$e \$i \$o \$u) includes: self asLowercase

or

isVowel

^'aeiou' includes: self asLowercase

A 5.10 Variables

Idiom V.1: Explicit Default Initialization

To initialize selected variables to default values, create a new method including initialize.

Example: If variable accounts is an OrderedCollection, include

accounts := OrderedCollection new

or a similar statement in the initialization method.

Idiom V.2: Lazy Initialization

If the value of a variable is not immediately required or may never be required, don't initialize it when the object is created. Instead, define a getter method that returns the value of the variable if the variable is not nil and assigns and returns the default value if the variable is nil.

Example: Accessing methods for widget aspects created with *Define* use lazy initialization.

Idiom V.3: Default Value Method

To make initialization more flexible, create a default method that calculates and returns the default value.

Examples: Abstract class Filename uses message defaultClass to determine which of its concrete subclasses will handle messages addressed to Filename objects. Class String uses the same principle. Changing the default then requires changing only the getter method.

Idiom V.4: Constant Method

To provide access to significant class values or to values shared across all instances, create a class variable or a class method that returns the value.

Examples: Class Float has method Pi that returns the value of π . Class ColorValue has class variables representing the most frequently used color values and class methods that return them.

Idiom V.5: Direct Variable Access

For better readability, access variables directly. Advantages: faster and protects encapsulation.

Examples: name := aString instead of self name: aString and width printString instead of self width printString.

Idiom V.6: Indirect Variable Access

For greater flexibility, access variables by accessing methods. Advantages: Provides a central place where change of variable occurs (easier to monitor), makes it easier to change internal class implementation.

Disadvantage: Breaks encapsulation (other objects can access variables), negligible message send overhead.

Example: Lazy initialization (Idiom V.2) requires that the variable be always accessed by an accessing method.

Note: The opinion on the use of *Direct* versus *Indirect* access varies from one expert to another.

Idiom V.7: Safe Collection Access

When an object contains a collection, access its elements via enumeration rather than by accessing the collection directly.

Example: Assume that class `Account` contains variable `transactions`. Method `transactionsDo: aBlock` allows flexible access to all transactions without allowing the accessor to change the value of individual transactions which is what would happen if you provided accessor method `transactions`.

Idiom V.8: Caching Temporary Variable

Use temporary variable to preserve a calculated object used repeatedly in a method or a block. If the calculation is lengthy or repeated many times, this will improve execution speed.

Example: Replace

```
department employeesDo:
    [:employee| Transcript show: 'Department ', department name, ' Employee ',
                                employee name; cr]

with
| label |
label := 'Department, department name, 'Employee '.
department employeesDo: [:employee| Transcript show: label, employee name; cr]
```

Idiom V.9: Explaining Temporary Variable

When a calculation requires a complex expression, save its parts in well-named temporary variables to make the code more readable.

Example: Replace

```
tax := ((incomes inject: 0 into: [:income :sum| sum + income]) -
        (expenditures inject: 0 into: [:expenditure :sum| sum + expenditure]) -
        (donations max: (donations - minDonations) * donationsRate)) * taxRate

with
| totalIncome totalExpenditures deductibleDonations |
totalIncome := incomes inject: 0 into: [:income :sum| sum + income].
totalExpenditures := expenditures inject: 0 into: [:expenditure :sum| sum + expenditure].
deductibleDonations := donations max: (donations - minDonations) * donationsRate.
tax := (totalIncome - totalExpenditures - deductibleDonations) * taxRate
```

Idiom V.10: Object Sharing Instance Variable

When several methods in the same class need the same object, save it in an instance variable instead of passing it as method argument.

Idiom V.11: Concatenating Stream

When concatenating several collections (usually `String` objects) use a `WriteStream` and its `contents` method.

Example: The `printString:` method defined in `Object` creates a `WriteStream`, passes it as an argument to `printOn:`, and returns the contents.

Idiom V.12: Cascading

When sending a series of messages to the same receiver, use cascading. Emphasize the receiver by indentation to improve readability.

Example:

```
self builder componentAt: #save
    lookPreferences foregroundColor: color;
    enable;
    beVisible
```

Idiom V.13: Adding, removing, and yourself

Adding and removing methods in collection and stream classes return the object being added or removed rather than the modified receiver. Use message `yourself` to return the modified receiver.

Example: `newCollection := oldCollection add: 13; yourself`

Appendix 6 - Projects

Overview

This appendix contains descriptions of many Smalltalk projects. Some are very detailed, others are just ideas that require some creative thought to arrive at a detailed specification. Some of the projects are easy, others are difficult.

A.6.1 1. When delivering a VisualWorks product to a customer, you might want to make it very difficult for the client to make sense out of your code - for competitive reasons. One strategy that has been suggested is to change the names of all your classes and methods to some meaningless sequences of characters. Develop a new class called **Srambler** which will do some or all of the following with each of the user-specified classes: Delete all comments (class and methods), rename the class and all its categories and methods to some random characters of strings. Note that the names must be changed wherever they are used, at all places in the library. Implementing this problem in a fully automated way is impossible because of polymorphism. You will thus have to implement the class so that the user provides some interactive help in the scrambling process.

2. 4.5 Dental Office

4.5.1. System Description

We are to implement a simple implementation of software required by a small dental office. The program will keep records on individual patients including their bills, the schedule of patient appointments, information on staff including their schedules and paychecks, and keep track of dental material and potential suppliers. In more detail, the information to be kept includes the following:

- Patient record
 - Name (first, middle, initial)
 - Address (mailbox, street, city, and postal code)
 - Phone number
 - List of appointments (date, time, task, bill, payment received)
 - Dental record (tooth-by-tooth: filled, extracted, nothing)
 - Amount currently due
 - Notes
- Staff record
 - Name (first, middle, initial)
 - Address (mailbox, street, city, and postal code)
 - Phone number
 - List of shifts (dates)
 - Hourly wage
- Material
 - Item name
 - List of suppliers
- Supplier
 - Name of company
 - Address
 - Phone
 - List of materials supplied with prices

We want to be able to perform the following tasks:

- Patient record
 - Add
 - Delete
 - Display
 - Edit
 - Print
 - Create appointment
 - Cancel appointment
 - Change appointment
 - Print information about appointment
- Staff
 - Add
 - Delete
 - Display
 - Edit
 - Print
 - Schedule (assign, change, cancel)
 - View shifts for current month
 - Print paycheck
- Material
 - Add
 - Delete
 - Display
 - Edit
 - Print
 - View all potential suppliers
 - Create order
 - Print order
- Supplier
 - Add
 - Delete
 - Display
 - Edit
 - Print

The Context Diagram is as in Figure 4.8.

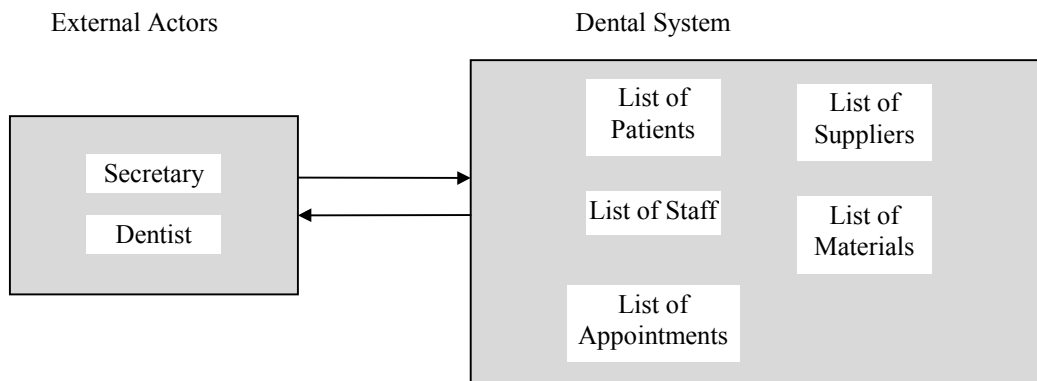


Figure 4.8. Context Diagram of Dental Office.

Our next task is to find the scenarios. Because this system is much more complicated than the library catalog, there will be many scenarios and we will thus restrict ourselves to a few examples only.

Scenario 1: Adding a new patient

The operator (secretary or dentist) selects the *New patient* task from the main window of the user interface, the system opens a window with fields for all required patient information, the operator enters all required information and clicks *Accept*, the system stores the new patient record.

Scenario 2: Making an appointment

The operator selects the patient from the list and the *Open patient record* task in the main window, the system opens a window with the patient record, the operator selects the *Appointment* task, the system opens an appointment window, the operator fills in the information, clicks *Accept* in the appointment window which closes, clicks *Accept* in the Patient record window which closes, the system stores the information.

Scenario 3: Creating and printing a material purchase order

The operator selects the *Create purchase order* task in the main window, the system opens a window with a list of materials, the operator selects a material, the system displays a list of suppliers of the selected material, the operator selects one, completes the *Quantity* field, clicks *Accept and Print*, the system saves the order and prints it.

This time, we will only show the main user interface window and leave it to you to decide which other windows are required and to design them. The main window is shown in Figure 4.9.

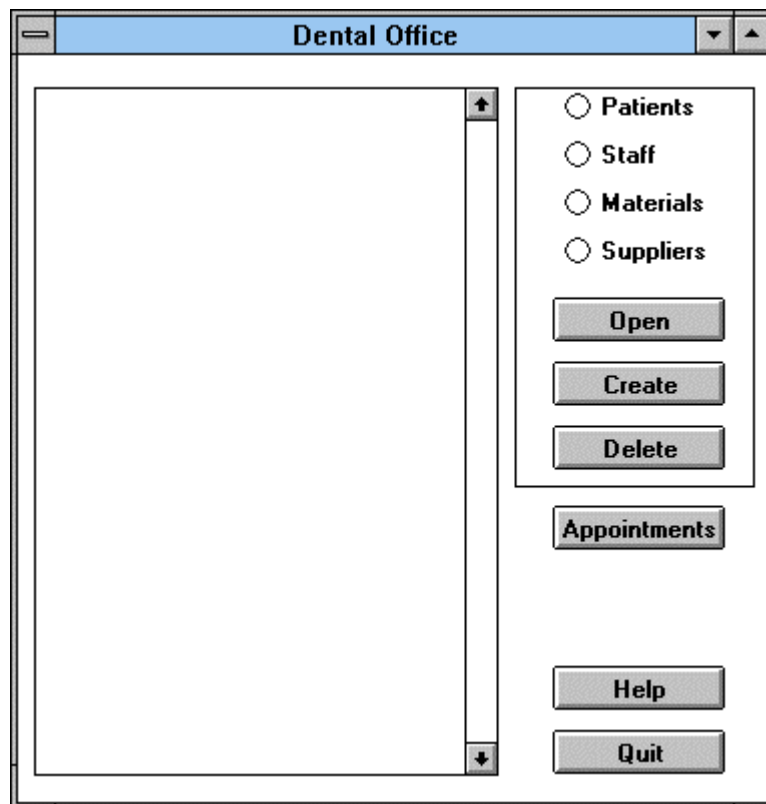


Figure 4.9. Main window of Dental Office.

3.

4.3 Computerized Library Catalog

4.3.1 System Description

We are to develop a computerized catalog for keeping track of books and borrowers. The program will allow authorized personnel (referred to as librarians) to perform the following tasks:

- Book
 - Add a new book to the catalog
 - Modify an existing book
 - Delete a book
 - Sign a book out to a borrower
 - Sign a returned book in
 - Search for a book in the catalog
- Borrower
 - Add a new borrower
 - Edit information about an existing borrower
 - Delete a borrower
 - Search for a borrower

The information held about books and records is as follows:

- Book
 - One or more authors identified by first, middle, and last names
 - Title
 - Publisher name
 - Year of publication
 - In or out of library
- Borrower
 - First name, middle name, and last name
 - Address consisting of street name and number, city, telephone number
 - List of borrowed books with dates when the books were borrowed

The specification assumes the user interfaces shown in Figures 4.5, 4.6, and 4.7.

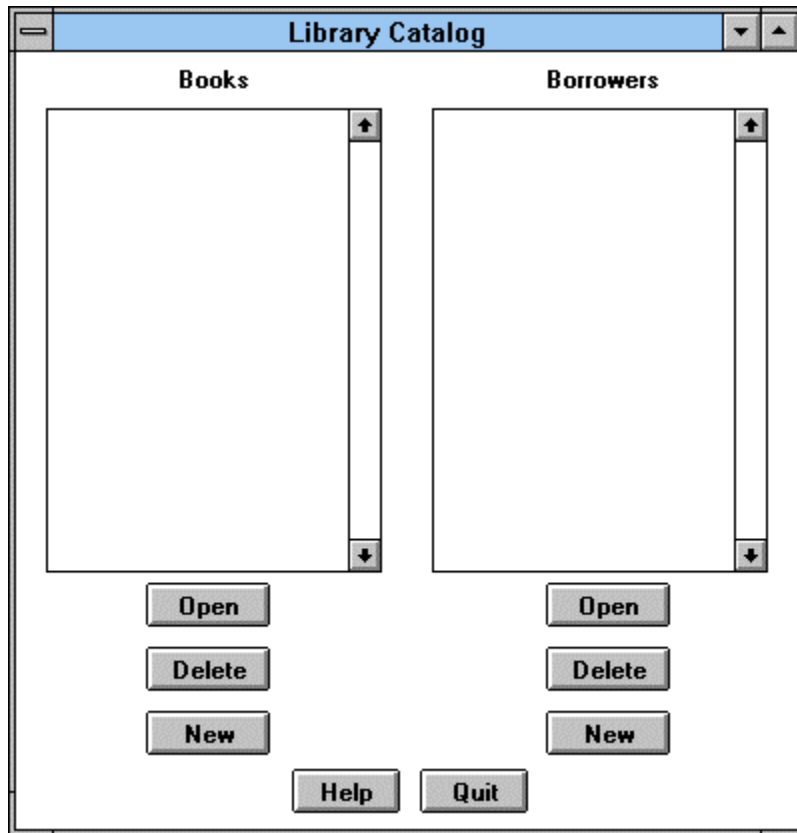
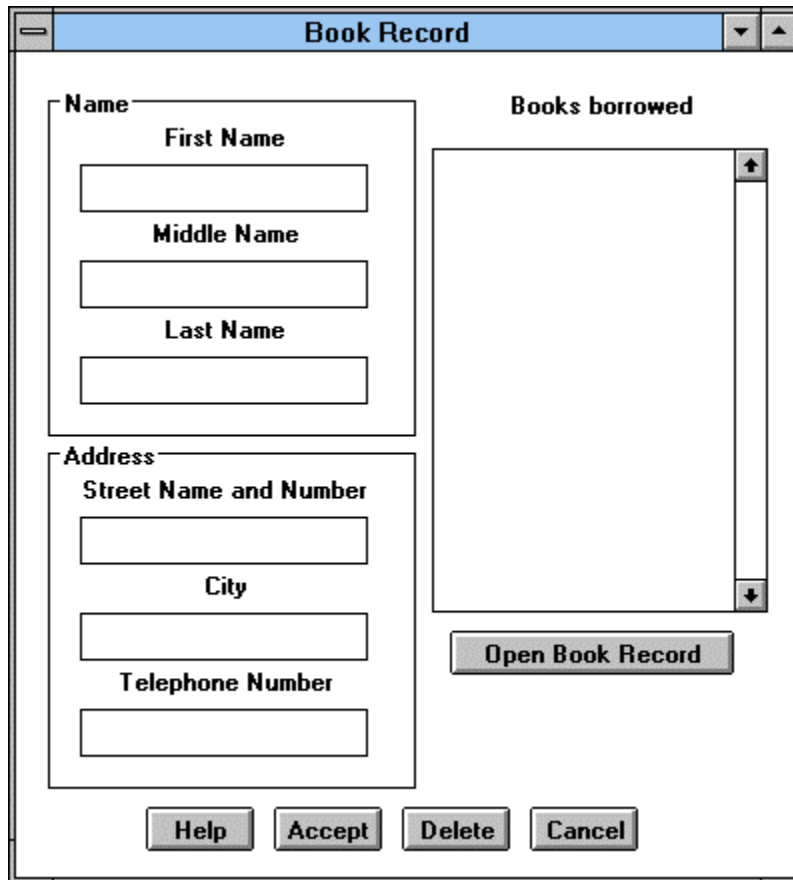
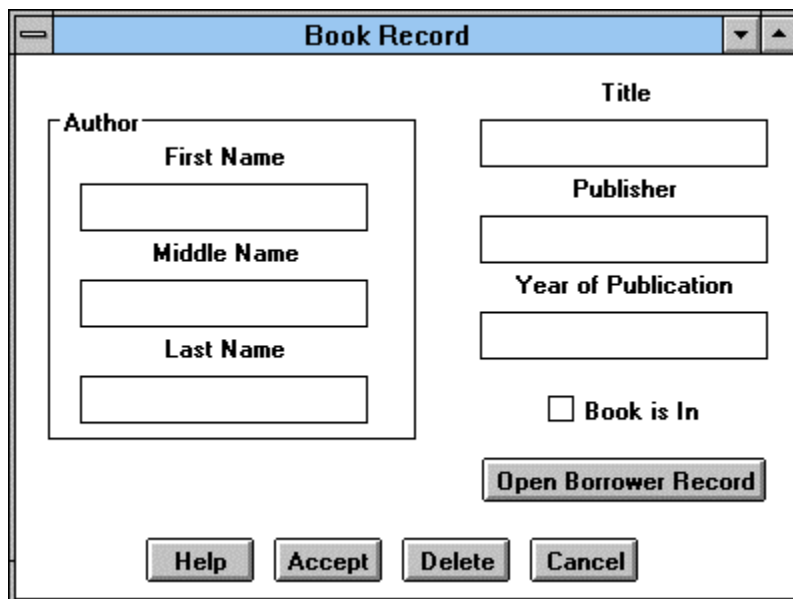


Figure 4.5. Main window of library catalog.



The 'Book Record' window features a blue title bar with standard window controls. It is divided into two main sections. The left section contains two grouped input areas: 'Name' with three stacked text boxes for 'First Name', 'Middle Name', and 'Last Name'; and 'Address' with three stacked text boxes for 'Street Name and Number', 'City', and 'Telephone Number'. The right section, titled 'Books borrowed', contains a large empty rectangular area with vertical scrollbars and a button labeled 'Open Book Record' positioned below it. At the bottom of the window, there is a row of four buttons: 'Help', 'Accept', 'Delete', and 'Cancel'.

Figure 4.6. Book information window.



The 'Borrower information' window has a blue title bar with standard window controls. It is divided into two main sections. The left section, titled 'Author', contains three stacked text boxes for 'First Name', 'Middle Name', and 'Last Name'. The right section contains three stacked text boxes for 'Title', 'Publisher', and 'Year of Publication'. Below these is a checkbox labeled 'Book is In'. A button labeled 'Open Borrower Record' is located below the checkbox. At the bottom of the window, there is a row of four buttons: 'Help', 'Accept', 'Delete', and 'Cancel'.

Figure 4.7. Borrower information window.

The following are some of the main usage scenarios:

Scenario 1: Librarian opens the application.

High level conversation:

1. *User* types a Smalltalk expression and executes it.
2. *System* opens a password window
3. *User* types password.
4. If the correct password is entered, *System* opens user interface shown in Figure 4.1; a notifier window opens otherwise.

Scenario 2: Librarian adds a new book.

High level conversation:

1. *User* clicks *New book* in the main window.
2. *System* opens a form window providing text fields for all required information. We will call this window the book form.
3. *User* enters at least author name and book title and indicates end of task by clicking *Accept* in the form window.
4. *System* closes form window and updates book catalog.

Scenario 3: Librarian modifies an existing book record..

High level conversation:

1. *User* selects the desired book in the catalog and clicks *Open*.
2. *System* opens a book form with current information about the book.
3. *User* edits the form and indicates end of task by clicking *Accept* in.
4. *System* closes form window and updates book catalog.

Scenario 4: Librarian deletes a book.

High level conversation:

1. *User* selects the desired book in the catalog and clicks *Delete*.
2. *System* opens window requesting confirmation that the book is to be deleted.
3. *System* closes confirmation window and updates book catalog if the user confirmed.

Scenario 5: Librarian signs a book out to a borrower.

1. *User* selects the desired book in the catalog.
2. *User* selects a borrower.
3. *User* clicks *Borrow book*.
4. *System* updates book and borrower catalogs.

Scenario 6: Librarian signs a book in.

High level conversation:

1. *User* selects a book.
2. *User* selects a borrower.
3. *User* clicks *Return book*.
4. *System* updates book and borrower information.

Scenario 7: Librarian adds a new borrower.

High level conversation:

1. *User* clicks *New borrower* in the main window.
2. *System* opens a form window providing text fields for all required information. We will call this window the borrower form.
3. *User* enters at least borrower name and address and indicates end of task by clicking *Accept* in the form window.
4. *System* closes form window and updates borrower catalog.

Scenario 8: Librarian modifies an existing borrower record.

High level conversation:

1. *User* selects borrower in the catalog and clicks *Open*.
2. *System* opens a book form with current information about the book.
3. *User* edits the form and indicates end of task by clicking *Accept* in.
4. *System* closes form window and updates borrower catalog.

Scenario 9: Librarian deletes a borrower.

High level conversation:

1. *User* selects borrower and clicks *Delete*.
2. *System* opens window requesting confirmation that the borrower is to be deleted.
3. *System* closes confirmation window and updates borrower catalog if the user confirmed.

Scenario 10: Librarian requests help information.

High level conversation:

- *User* clicks *Help*.
- *System* opens help window.
- *User* clicks *Close* in the help window.
- *System* closes help window.

This scenario is so obvious that it is questionable whether it should be included. We will not include such trivial scenaria in the future.

A variety of other important scenaria exist and we leave them as exercises.

The specification and scenaria that we just read contain several new terms and it will be useful to create a dictionary with their exact definitions. *We will add a Term Dictionary as a new deliverable for Specification of Requirements.* The dictionary for our problem is as follows:

- *Book form* - dialog window in Figure 4.6. Contains book information and additional control buttons.
- *Borrower form* - dialog window in Figure 4.7. Contains borrower information and additional control buttons.

(A dialog window is a window that must be completed and closed before another window can be accessed.)

Context Diagram

The last task is to draw a diagram showing the main components and deciding which of them are a part of the application, and which external actors. From the description, we conclude that there is a single external actor - the librarian - and that the system to be developed consists of a user interface, an application model, a catalog of books, and a catalog of borrowers.

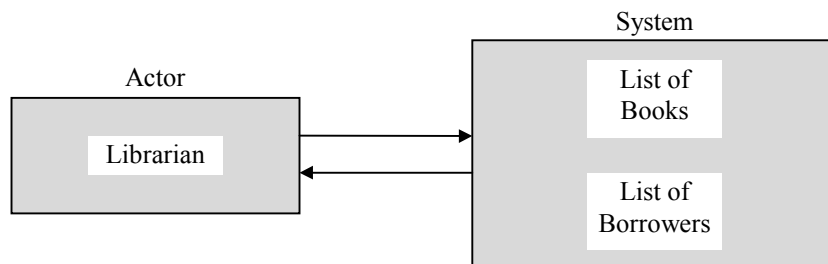


Figure 4.4. Context Diagram showing readily identifiable parts of Library Catalog and external actors.

This completes System Description and we proceed to Preliminary design.

4.3.2 Preliminary design

The steps in Preliminary Design are

1. Identify candidate classes and create CRC cards with their brief descriptions.
2. Identify responsibilities of candidate classes and record them on their CRC cards.
3. For each class, identify collaborator classes required to fulfill this class's behaviors and record them on the CRC card.
4. Verify completeness of CRC cards by walkthroughs, checking that the identified classes and their behaviors can implement all scenarios obtained during the System Description step.

Identify candidate classes and their responsibilities

This is the most difficult part of the whole procedure and the commonly proposed guideline for finding the classes is to read the description carefully, looking for nouns as possible class candidates. In our relatively simple example, the obvious candidates are the windows and the other components of the user interface, the list of books, and the list of borrowers.

One would expect that *windows and their components* are already in the Smalltalk library and this is indeed the case. We can thus eliminate them from our consideration since we are only interested in the *new* classes that we will have to create.

The list of books and the list of borrowers are both lists, collections of objects. Collections of objects appear in almost every application and we would again expect that suitable collection classes are already in the Smalltalk library. This is indeed so and we thus don't have to include them in our design.

The only remaining objects therefore are books and borrowers. A book object is basically an information object and its intelligence is limited to knowing certain information, being able to provide it when requested (for example when a window needs to display it), and being able to change it when requested (when the librarian enters new information). A borrower object is another information object capable of holding and accessing information; only the type of information it holds is different. A reasonable conclusion is that we need to declare two new classes - *Book* and *Borrower*. This, however, probably is not the best solution for two reasons: The first is that both of these objects contain a bit too much information, the second is that the information held by both *Book* and *Borrower* contains smaller and logically self-contained concepts - person information and address information - and that some of this information - namely person information - is, in fact, shared by both. We conclude that a more elegant solution is to define the following classes:

Address: Contains and can access information about street name and number and city.

Book: Contains and can access information about book consisting of author, title, publisher, and isIn information.

Borrower: Contains and can access information about borrower consisting of borrower, address, telephone number, and list of borrowed books.

Person: Contains and can access the first, middle, and last name of a person.

Besides greater simplicity of these classes, the advantage of this approach is that we may be able to reuse them in another application. After all, persons and addresses are very common. We summarize our decisions on CRC cards as in

Book: I hold information about a book in the library - its author, title, publisher, year of publication, and status (in or out).

Responsibilities

Get/return author information

Get/return title information

Get/return publisher information

Get/return year information

Get/return isIn information

Collaborators

Person

We leave it to you to complete CRC cards for the remaining classes as an exercise.

Verify completeness of class list by scenaria walk throughs

Scenario 1: *Librarian opens the application.*

The librarian types a Smalltalk expression and executes it. The application open the user interface

Scenario 2: *Librarian adds a new book to the catalog.*

Librarian selects the *New* task from the main window, a window appears providing text fields for all required information. The librarian enters at least author name and book title, indicates end of task by clicking *Accept* in the main window, the application closes the window and updates the internal catalog.

Assuming that we have a working user interface, clicking the appropriate buttons and entering information stores the information in the object associated with the interface. Clicking the *Accept* button then requests a new *Book* object from the *Book* class, transfers the collected information to it, and adds it to the book catalog object. The scenario thus seems executable. However, there is a small gap here - we have a *Book* class and the *Person* class defining its major component, we also have classes implementing windows and their 'widgets' (buttons, labels, lists, input fields, and so on) - but we don't have a class to tie the user interface to the book list, borrower list, book objects, and borrower objects. In Smalltalk terminology, we have the *UI* (user interface) classes and the *domain* classes describing the objects in our problem domain, but we don't have the *application model* (Figure 4.5).

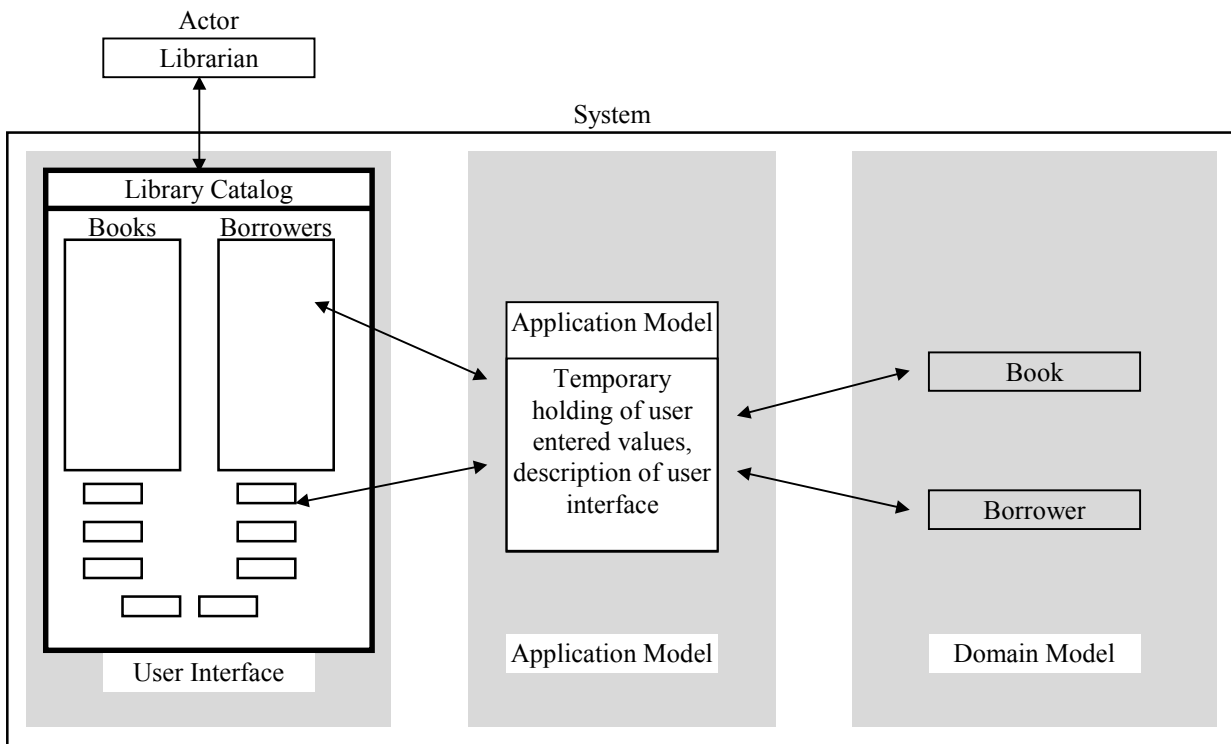


Figure 4.6. Smalltalk applications consist of a user interface, a domain model, and an application model.

We will thus add another class called **LibraryCatalog**. Its responsibilities will include

- holding the description of the user interface (how the window and UI widget objects and laid out and how they communicate the information that they display or gather)
- holding information entered by the librarian before it is accepted and transferred to domain objects.

We leave it to you to verify that our classes can handle all the remaining scenaria.

4.3.3 Design Refinement

In this step, we must decide whether some of our classes are sufficiently similar in their behavior to warrant declaring a superclass with the shared behavior, and we must provide details for the behaviors that we identified in the previous step. In our case, the classes that we identified don't share any behaviors and there is no justification for any new superclasses. All classes, with the exception of **LibraryCatalog** have no shared behavior with any existing classes in the Smalltalk library and we will thus define them as subclasses of **Object**. Class **LibraryCatalog**, on the other hand, contains mainly behavior that is shared by all application models - it holds the description of the user interface and provides access to domain objects - and it is not surprising that **VisualWorks** contains an abstract class providing much of this behavior. This class is called **ApplicationModel** and **LibraryCatalog** will thus be a subclass of **ApplicationModel** (Figure 4.7).

Class name: LibraryCatalog	Superclass: ApplicationModel
Responsibilities	Collaborator classes
Hold description of user interface	
Hold book information until accepted	
Hold borrower information until accepted	

Figure 4.7. CRC card of class `LibraryCatalog`.

The last step is to decide whether we should refine some of the responsibilities that we identified in the previous steps. Since our domain classes are trivial information objects, all their simple behaviors are accessing behaviors that don't require any elaboration. Since we cannot implement our classes, we are thus finished.

Example: Puzzle-a-day program

use sorted dictionary but implement it as sorted collection.

Chapter projects

All project classes are available in the file-in code for this book.

Project 1: Function calculator

Re-implement Project 1 from Chapter 5 using the user interface in Figure 3.30. The project is available as Project4 in the fileout.

Figure 3.30. User interface for Project 1.

Project 2: The Hangman game?

Implement the Hangman game with the interface in Figure 3.31. Use the Divider and Region widgets from the canvas, beVisible and beInvisible (?) messages. Explain game. Read Chapter Input Fields, etc. first

Project for Chapter 8

Implement the bank simulation project used as an example in Chapter 2.

Projects for chapter 13

- 1.
 2. Elevators
 3. Port
 4. Bank
1. Extend the shopping minder from Chapter 9 into the following application: The user interface consists of a single window with two single-selection list widgets and several buttons. The list on the left contains all available items and their unit prices, the list on the right contains items that the user selected and their counts. The buttons include: *Add* - asks the user for a new item name and adds it to the list of available items or increments the item's count if it is already in the list, *Get* - adds the item selected in the left list and adds it to the right list, *Delete* - subtracts one from the number of occurrences of the item selected in the right list, *Print* - prints information in the right list in the Transcript, *Close* - closes the application.

Appendix 7 - Smalltalk Syntax

Overview

This appendix presents the formal syntax of Smalltalk grammar - a collection of formal prescriptions for forming valid Smalltalk expressions and reading Smalltalk code correctly. The rules have been copied from the 1996 Smalltalk Draft with some modifications reflecting changes that have been decided by the committee in the meantime. The Draft contains a wealth of very interesting information going well beyond the listing of Smalltalk syntax and we recommend it for reading. With respect to the following listing, note that a draft is not a final standard and the definitions are still open to change.

A.7.1 Syntax rules

The following rules, copied and slightly corrected from the Smalltalk Draft, use the Backus-Naur-Form (BNF) notation with a non-terminal member of the grammar (a concept being defined) on the left hand side and its definition on the right hand side of the ::= symbol. We also use

	to represent logic or
+	to represent one or more occurrences
*	to represent zero or more occurrences
..	to represent all symbols from the symbol on the left to the symbol on the right
[]	to represent optional symbols
()	to represent grouping
<>	to denote a non-terminal (a syntactic concept being defined)
' '	surrounds a character
" "	surrounds a textual comment

Although the following list of rules appears long, most of the rules are auxiliary and trivial only a relatively small number of rules define substantial concepts – essentially the Expressions and Method Definition. Compared to most other languages, Smalltalk's grammar is very simple.

Lexical primitives

<character>	::=	"Any character in the 7-bit ASCII character set" "Any character in the UNICODE character set"
<nonPrintingCharacter>	::=	"Any UNICODE character interpreted as white space"
<digit>	::=	'0' .. '9'
<digits>	::=	digit+
<uppercaseAlphabetic>	::=	'A' .. 'Z' "Any UNICODE character interpreted as uppercase alphabetic"
<lowercaseAlphabetic>	::=	'a' .. 'z' "Any UNICODE character interpreted as lowercase alphabetic"
<lexicalNumber>	::=	<digit> (['.'] (<letter> ['-'] <digit>)) *
<radixDigits>	::=	(<digit> <uppercaseAlphabetic>) +
<number>	::=	digits ['r' <radixDigits> '.' <digits> ['e' ['-'] <digits>]
<stringDelimiter>	::=	" "
<nonStringDelimiter>	::=	"<character> that is not a <stringDelimiter>"
<string>	::=	<stringDelimiter> (<nonStringDelimiter> <stringDelimiter> <stringDelimiter>) * <stringDelimiter>
<nonCaseLetter>	::=	'_' "Any UNICODE character interpreted as non-case letter."
<letter>	::=	<uppercaseAlphabetic> <lowercaseAlphabetic> <nonCaseLetter>
<identifier>	::=	<letter> (<letter> <digit>) *
<keyword>	::=	<identifier> ':'
<commentDelimiter>	::=	' " '
<nonCommentDelimiter>	::=	<character> where <character> is not a <commentDelimiter>

<comment>	::=	<commentDelimiter> <nonCommentDelimiter> <commentDelimiter>
<binaryCharacter>	::=	'+'/'\"/>

Atomic Terms

<literal>	::=	<numberConstant> <stringConstant> <characterConstant> <symbolConstant> <arrayConstant> <identifier>
<numberConstant>	::=	['-'] <number>
<stringConstant>	::=	<string>
<characterConstant>	::=	'\$' <character>
<hash>	::=	'#'
<symbol>	::=	<identifier> <binarySelector> <keyword>+
<literalSelector>	::=	<hash> <symbol>
<literalSymbol>	::=	<hash> <string>
<symbolConstant>	::=	<literalSelector> <literalSymbol>
<arrayConstant>	::=	<hash> '(' <literal>* ')'
<variableName>	::=	<identifier>

Expressions

<primary>	::=	<identifier> <literal> <blockConstructor> <subExpression>
<unarySelector>	::=	<identifier>
<unaryMessage>	::=	<unarySelector>
<binaryMessage>	::=	<binarySelector> <primary> <unaryMessage>*
<keywordMessage>	::=	(<keyword> <primary> <unaryMessage>* <binaryMessage>*)+
<cascadedMessage>	::=	(';' (<unaryMessage> <binaryMessage> <keywordMessage>))*
<messages>	::=	<unaryMessage>+ <binaryMessage>* [<keywordMessage>] <binaryMessage>+ [<keywordMessage>] <keywordMessage>
<restOfExpression>	::=	[<messages> <cascadedMessages>]
<expression>	::=	<variableName> (<assignmentOperator> <expression> <restOfExpression>) <variableName> <assignmentOperator> <expression> <primary> <restOfExpression> 'super' <messages> <cascadedMessages>
<temporaries>	::=	['' <temporaryList> ']' '']
<temporaryList>	::=	<declaredVariableName>*
<declaredVariableName>	::=	<variableName>
<blockConstructor>	::=	['' <blockDeclarations> <statements> '']
<blockDeclarations>	::=	<temporaries> <blockArgument>+ ']' [<temporaries>]
<blockArgument>	::=	':' <variableName>
<statements>	::=	[<returnOperator> <expression> ['.'] <expression> ['.'] <statements>]]

Note: The definition of <blockDeclarations> has been corrected with respect to the Draft.

Method Definition

<method>	::=	<messagePattern> <temporaries> <statements>
<messagePattern>	::=	<keywordPattern> <binaryPattern> <unaryPattern>
<keywordPattern>	::=	(<keyword> <identifier>)+
<binaryPattern>	::=	<binarySelector> <identifier>
<unaryPattern>	::=	<identifier>

Note: The last four rules are not included in the Draft which misses a definition of <messagePattern>. The four rules correspond to the heading of a keyword, binary, and unary method respectively.

Conclusion

Compared to almost all other languages, Smalltalk syntax is extremely simple because it can be reduced to the following simple points: The definition of three types of messages, the rule that a message has a receiver, the rule about combining messages into expressions and messages into statements, cascading, and the rule for the formation of literals, blocks, and methods. Unlike most other languages, Smalltalk does not have almost any reserved words with special meaning and disallowed in source code.

Appendix 8 - Tidbits

Overview

This appendix presents several unrelated topics that are interesting enough to be included in the book but that did not fit into the main chapters or other appendices. Consider this appendix to be a dessert.

The topics included here are material on the processing of keyboard events and how you can take advantage of it to define or redefine the behavior of selected keys, an introduction to text and fonts, a presentation of drag-and-drop in GUIs, an introduction to garbage collection, and an introduction to the classical model of the Virtual Machine.

A.8.1 Keyboard input

In this section, we will explore the basics of Smalltalk's processing of keyboard keys and how we can take advantage of the process.

How it works

When you press a key on the keyboard and a Text Editor or Transcript is in control, Smalltalk first checks whether the key has a special meaning. If it does (as an example, if you press the <Delete> key), it executes the appropriate operation, if it does not, it just adds the corresponding character to the output. We will now explore how this process works and how you can take advantage of it.

The processing of user input is of course performed by the controller of the Text Editor which is an instance of `ParagraphEditor`. Processing of keyboard keys occurs in the `editing` protocol which defines response to a variety of special keys and we will use the method defining response to one of the less offensive keys to examine how the process works.

The method that we selected is `backspaceKey: aCharEvent` and we inserted `self halt` at its beginning to intercept the activation of the <Backspace> key and to open a Debugger. We then typed some text and pressed <Backspace>. The Debugger opened, and we observed the following (Figure A.8.1).

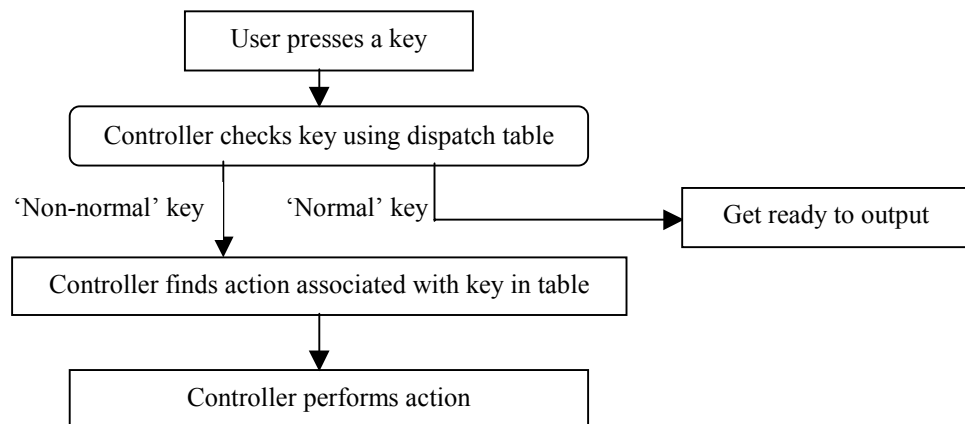


Figure A.8.1. Keyboard processing sequence.

The existing implementation of the `ParagraphEditor` is based on polling. When an activity occurs, the `controlActivity` method in the polling loop execute two messages in the order `pressKeyboard`, and `processMouseButtons`. We will focus on `pressKeyboard` and leave `processMouseButtons` as an exercise.

Method `processKeyboard` first checks whether a key has indeed been pressed. If so, it sends message `readKeyboard`. This message check whether the character is a 'normal' character or not (<Backspace>, up or down arrow, <ESC> followed by another character, and so on). Normal characters are

added to a stream and displayed in the text view, whereas non-normal characters are 'dispatched' for further processing. This processing consists of checking ParagraphEditor's dispatch table, an instance of DispatchTable, whose function is to associate keyboard keys and their combinations with methods. A non-normal character or its combination with other characters results in the lookup of the appropriate method, which is then executed. In our case, we pressed <Backspace>, its associated method is backspaceKey:, and this method is executed next.

The essence of keyboard processing, from the developer's point of view, is the dispatch table associated with ParagraphEditor. To take advantage of the keyboard response process, we must thus understand the contents of the dispatch table. The dispatch table is associated with the ParagraphEditor class, and the initialization of the dispatch table thus happens in a *class* initialization method. This method is called initializeDispatchTable and the representative pieces of its definition are as follows:

initializeDispatchTable

```
Keyboard := DispatchTable new.
"First define default behavior."
Keyboard defaultForCharacters: #normalCharacterKey:.
Keyboard defaultForNonCharacters: #ignoreInputKey:.
"Now define non-default behavior by associating keys with methods for keys that have
distinguished behavior."
"First behaviors for special keys on the keyboard."
Keyboard bindValue: #deleteKey: to: Cut.    "Execute method deleteKey: if Cut (?) is pressed."
Keyboard bindValue: #pasteKey: to: Paste.
Keyboard bindValue: #backspaceKey: to: BS.
Keyboard bindValue: #backWordKey: to: Ctrlw.
"First special Ctrl keys for creating frequently needed text."
Keyboard bindValue: #displayIfTrueKey: to: Ctrlt.    "Hot key for inserting ifTrue."
Keyboard bindValue: #displayIfFalseKey: to: Ctrlf.
Keyboard bindValue: #displayDateKey: to: Ctrlld.    "An example of what you might do."
Keyboard bindValue: #displayColonEqualKey: to: Ctrlg.
"Now some text editing hot keys."
Keyboard bindValue: #findKey: to: Ctrlfs.
Keyboard bindValue: #findDialogKey: to: Ctrlfa.
Keyboard bindValue: #replaceKey: to: Ctrlr.
Keyboard bindValue: #replaceDialogKey: to: Ctrlre.
"Now keys such as Enter, up and down arrow keys, etc."
Keyboard bindValue: #displayCRKey: to: #Enter.
Keyboard bindValue: #cursorUpKey: to: #Up.
etc.
"Next behaviors for special key combinations of a character preceded by ESC."
'<""[{' do: "Any of these preceded by ESC puts 'brackets' around the selected text."
    [:char |
    Keyboard
        bindValue: #encloseKey:
        to: ESC
        followedBy: char].
'sSuUbBilx+-' do: "This one changes the style of the selected text."
    [:char |
    Keyboard
        bindValue: #changeEmphasisKey:
        to: ESC
        followedBy: char].
Keyboard
    "This one performs mini-formatting of code when you press ESC and then f."
    bindValue: #miniFormatKey:
    to: ESC
    followedBy: $f.
etc.
```

The only mystery in this definition is the meaning of symbols such as Ctrlt, Ctrlf, Cut, Paste, and BS. According to the way that these identifiers are written, they must be global variables (no, they are not of

global interest), or class variables, or keys in a pool dictionary¹. It turns out that they are keys in the pool dictionary `TextConstants` because they are used by several other classes as well. When you inspect this dictionary, you will find that `Ctrl`, `Ctrlf`, `Cut`, `Paste`, and `BS` are among many characters that are assigned special names for ease of processing. As an example, `Ctrl`'s value is the character whose hexadecimal code is "16r0014", in other words, the character whose ASCII code is 14H, often referred to as *control t*².

Now that we fully understand how this works, we can explore how we can use it.

How we can take advantage of it

There are three ways in which we can use this knowledge: We can use the built-in behaviors, we can modify the key combinations, and we can define new behaviors.

Existing behaviors. Surprisingly, much of the built-in behavior is not documented in the User Guide. As an example, did you know that if you select text and then click `ESC` followed by `i`, the selection will be italicized? If you read the above method carefully and explored the behaviors implemented by the dispatch table, you now know. The following is a summary of most of the predefined behaviors and we urge you to explore the definition for the rest and test them in the Workspace.

Key or key combination	Brief description	Detailed description
Arrow keys		
<Ctrl> <a>	'Find' dialog	Same effect as <i>find</i> command in the <operate> menu.
<Ctrl> <d>	Display date	Display today's date at current cursor position.
<Ctrl> <e>	'Replace' dialog	Same effect as <i>replace</i> command in the <operate> menu.
<Ctrl> <j>	Paste	Same effect as <i>paste</i> command in the <operate> menu.
<Ctrl> <h>	Erase character	Same effect as <i>delete</i> command in the <operate> menu.
<Ctrl> <w>	Erase word	Erase word preceding current position of cursor.
<Ctrl> <i>	Tab	Same effect as <Tab> key.
<Ctrl> <f>	ifFalse:	Enter ifFalse: at current cursor position.
<Ctrl> <g>	:= (get)	Enter := at current cursor position.
<Ctrl> <t>	ifTrue:	Enter ifTrue: at current cursor position.
<Ctrl> ...	Control characters	According to the ASCII standard, control codes such as <Ctrl> <I> have functions such as Tab and others. See <code>TextConstants</code> and an ASCII table for definition.
<ESC> Tab	Select typed	Select (highlight) recently typed text.
<ESC> surround character	brackets toggle	Add or remove < around selected text. Similar effect also for <[> <{> <'> <(> <(> keys.
<ESC> <f>	mini-format	Format highlighted text.
<ESC> style character	change emphasis	Turn style of selection on or off. bold, <u> underline, <i> italic, <s> serif, <+> larger font, <-> smaller font, <x> emphasis at start of block. Lowercase letter turns emphasis on, uppercase letter turns it off.
F1 function key	Select typed	Select (highlight) recently typed text. Same as <ESC> <tab>

Redefining keys. The next thing you can do is redefine the keys defining the existing behaviors. As an example, if you are used to Microsoft Word conventions, you might want to redefine the *Paste* key to be <Ctrl> <v> instead of <Ctrl> <j>. When you look at the definition of the dispatch table above, you will see that the paste key is defined by the `Paste` pool variable and to change it, you must change the value of `Paste` in the `TextConstants` pool dictionary. One way to do this is to execute

¹ They are not, obviously, classes.

² The first 32 ASCII codes do not have printable representation although some of them represent characters such as Tab that have effect on output. Because they have control functions, they are known as control characters and often referred to by letters of the alphabet: Code 1 is control a, code 2 is control b, etc.

TextConstants at: #Paste put: (TextConstants at: #Ctrlv)

After this, you can check that Paste in TextConstants has now the value of "16r0016". However, this is not the end because we must now reinitialize the ParagraphEditor with this new value by

ParagraphEditor initialize

Everything should now work. If you have an open Workspace, type something in, make a selection, copy it, and try <Ctrl> <v> to see if it performs the *Paste* operation. You may be surprised to find that it does not and the reason is that if the Workspace existed before you made the change, it still has the old setting associated with it. Open a new Workspace and try it, and this time it works.

Before we conclude this subsection, we must note that you cannot redefine <Ctrl> <c> because this key is permanently assigned to the *Break* function (open User Interrupt Exception) in the Virtual Machine.

New behaviors and keys. We can define new key behaviors in the same way as the old ones, as long as our new definition does not override existing ones. In this example, we will define a new hot key combination to display *self halt* at the cursor position because this is a frequently needed expression. We will assign this function to <Ctrl> <h> to make it easy to remember.

In our implementation, we will, of course, shamelessly steal from existing code (the term is 'reuse'). What we need to do is update the dispatch table and this must be done by editing its initialization method and then reinitializing the ParagraphEditor. When you analyze the definition of *initializeDispatchTable* above, you will find that we should model our addition after

```
Keyboard bindValue: #displayIfTrueKey: to: Ctrlt
```

and we thus add

```
Keyboard bindValue: #displaySelfHalt: to: Ctrlh
```

It now remains to add the new method *displaySelfHalt*: which we edit from the existing definition of *displayIfTrueKey*: as follows:

displaySelfHalt: aCharEvent

"Replace the current text selection with the string 'self halt'."

self appendToSelection: 'self halt'

After this, we reinitialize ParagraphEditor by

ParagraphEditor initialize

open a new Workspace, try <Ctrl> <h> - and it works! Unfortunately, there is one problem: <Ctrl> is already used for the *Backspace* character and when we redefined it, *Backspace* does not delete the previous character but inserts *self halt*. We suggest that you change the combination from <Ctrl> <h> to <Ctrl> for 'create break'. This works without problems.

In addition to binding an action to a single key, the dispatch table can also bind an action to a combination of two keys. The following statement from *initializeDispatchTable* shows how to do this:

```
Keyboard  
  bindValue: #selectCurrentTypeInKey:  
  to: ESC  
  followedBy: Tab.
```

Finally, a note on one keyboard feature that is implemented differently. As you know, the *Delete* key by default acts like the *Backspace* key, in other words, deletes the previous character. In most word

processors, such as Word, *Delete* remove the next character. This can be change by sending the following class message

LookPreferences deleteForward: true

Since this message changes a class variable and since values of class variables are persistent, this changes the behavior of your *Delete* key permanently, until you change it back again.

Main Lessons Learned

- Keyboard processing is handled by the controller class ParagraphEditor.
- ParagraphEditor checks each activated key and if the key (or a combination of this key and other keys) is in its dispatch table, it executes the associated action by performing the attached method.
- The dispatch table defines a number of useful undocumented behaviors.
- The method initializing the dispatch table can be edited to obtain new behaviors or to bind different keys to existing behaviors.
- The operation of <Ctrl> <c> is an exception because it is defined in the Virtual Machine and cannot be redefined.
- Class LookPreferences also plays a role in keyboard processing.

Exercises

1. Class DispatchTable is a very interesting class that is the basis of keyboard mapping. Write its short description.
2. The effect of different entries in ParagraphEditor's dispatch table can be grouped into several categories. List these categories and explain how they are implemented.
3. Classify entries in TextConstants into categories and give one example of each.
4. How is the change of emphasis implemented?
5. Redefine other shortcut keys including <Ctrl> <x> to *Delete*, and <Ctrl> <z> to *Undo*. Try whether you can redefine <Ctrl> <c> to *Copy*.
6. As we noted, <Ctrl> <d> prints today's date. Define an escape sequence <ESC> <d> to open a notifier window with today's date and current time.
7. Explain the operation of the binding of a block to a function key. (Hint: The table above includes an example.)
8. Reimplement Exercise 6 to use function key F2 instead of the combination <ESC> <d>.
9. What is the mechanism by which class LookPreferences changes the behavior of the *Delete* key?
10. Our extension of hot keys to include <Ctrl> <h> works in the Workspace but not in the Browser. Why? Correct this shortcoming.

A.8.2 Text and fonts

As we know, a Text object consists of a string and emphasis where emphasis assigns to each character in the text one or more symbols or an association, such as #bold, or #(#bold #underlined), or #color -> ColorValue red. The obvious question that arises is how is this information converted to a font, size, color, and additional information needed to display the text such as line spacing, number of pixels representing a tab, and so on. To understand the mechanism and to be able to understand it, we must now introduce class ComposedText and its relatives.

Class ComposedText and its relatives

For display, Text, must be converted to ComposedText as in

aText asComposedText

A `ComposedText` contains `Text` and has access to a dictionary of text styles via its `TextAttributes` component (Figure A.8.2). The `TextAttributes` object does not itself contain the style dictionary but has access to a `CharacterAttributes` object which contains this dictionary. (In addition to an instance of `CharacterAttributes`, `TextAttributes` also contains information such as how many pixels is a tab, how many pixels separate two lines, what is the alignment of the text – centered, left aligned, and so on – etc.). The style dictionary in `CharacterAttributes` translates individual emphasis symbols using blocks that modify the base (default) font associated with the `CharacterAttribute`. (This default font is an instance of `FontDescription`.) Depending on how many arguments the block has, the transformation is done by

- accessing the font only or
- the font and an argument that could prescribe, for example the size of the font, or font, argument, or
- a `FontPolicy` object which transforms a `FontDescription` to an available platform font `ImplementationFont`. Note that the value of the dictionary might also be an array of symbols, in which case the translation is performed again using the same dictionary.

When a `ComposedText` is asked to display itself, as in

a `ComposedText displayOn: aGraphicsContext`

it uses its `TextAttributes` to convert tabs to pixels, calculate line spacing, and so on, and performs the translation of the emphasis of its individual characters from symbols to fonts. Using `FontPolicy` it then translates the obtained desired font to the nearest available `ImplementationFont`.

Given these principles, we can now explain how to control the font and other visible aspects of the displayed text (Figure A.8.3). A `ComposedText` normally uses default system `TextAttributes` associated with default `CharacterAttributes` and its associated dictionary. For most applications, the default dictionary with its definition of `#color`, `#bold`, `#large`, `#small`, `#underlined`, and other text styles is sufficient. When it is not sufficient, we have the following two possibilities which can be combined:

- We can associate the `ComposedText` with its own `TextAttributes` object with its own parameters, including a new `CharacterAttributes` object with its own custom dictionary. This dictionary can define new emphasis symbols and associate each of them with a block that transform the base font as desired. The base font itself is the font returned by message `defaultQuery` sent internally to the `CharacterAttributes` object. Some of the parameters that we might want to define for our new symbols are color, size, font family, underlining, and so on.
- Another way to introduce new fonts is to redefine the base font by `defaultQuery: aFontDescription`. With this approach, the blocks that transform the base font when they implement emphases start from a non-default font.

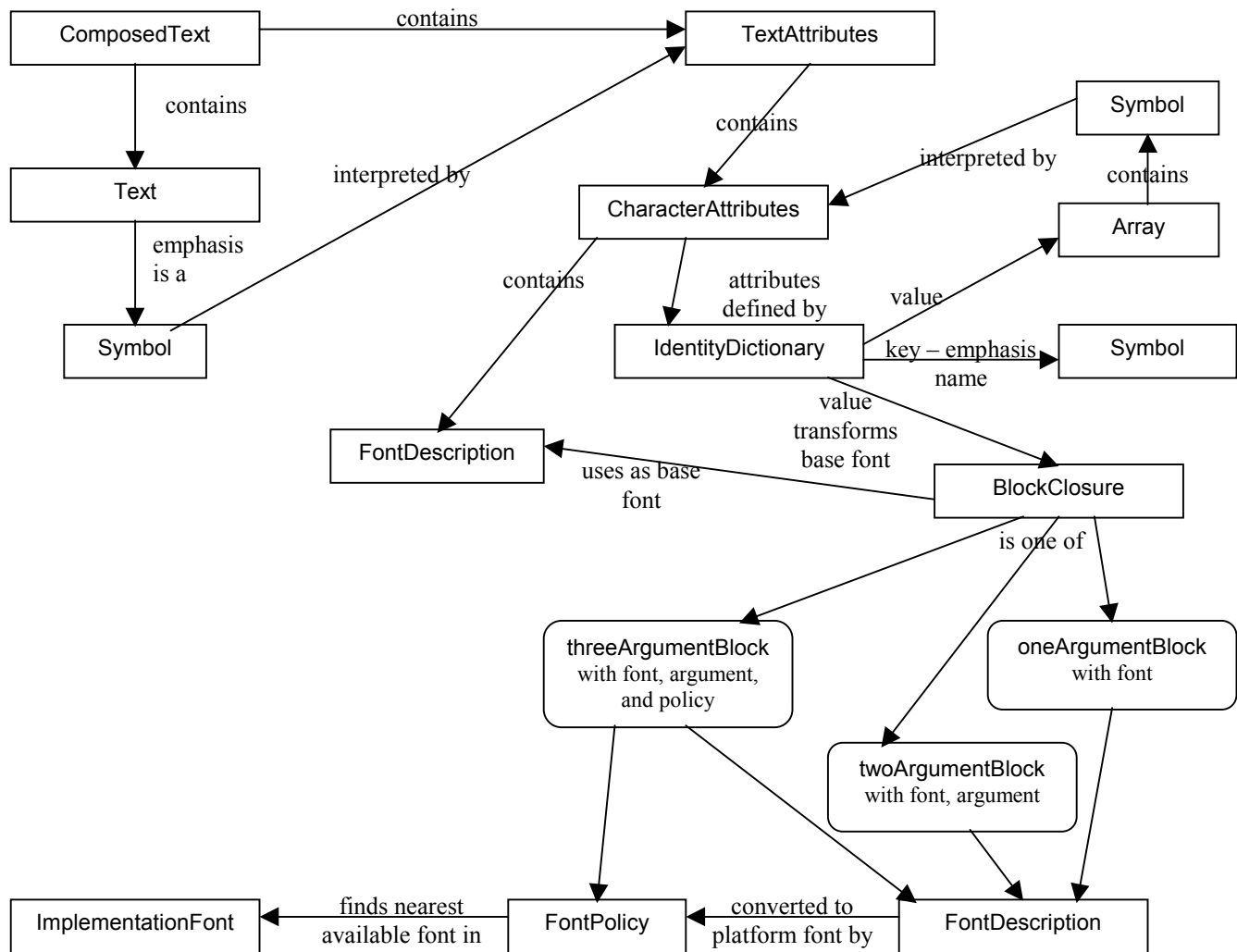


Figure A.8.2. Object diagram showing the participants in conversion of text symbols to fonts.

Figure A.8.3. Usage diagram showing the process of conversion of text symbols to fonts.

We will now give several examples showing these basic approaches to creating new fonts and their variations.

Example 1: Define new font color attributes – a problem requiring a one-argument emphasis block

Problem: Define new emphases **#red** and **#blue** that change font color.

Solution: The solution does not require a new kind of font (we are just modifying attributes of the existing font) and consists of the following steps:

1. Create a new **CharacterAttributes** object from the default **CharacterAttributes**.
2. Define its **defaultQuery** to return the default font.
3. Add the **symbol -> block** entry to the **CharacterAttributes** dictionary (which already contains all predefined default emphases).
4. Create a **TextAttributes** with the new **CharacterAttributes**.
5. Assign the new **TextAttributes** as the text's new text style.

The code fragment implementing this algorithm and displaying the text in the currently active window is as follows:

```
| composedText ca ta gc |
"Get text."
composedText:= 'A text' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #red put: [:fontDesc | fontDesc color:ColorValue red].
ca at: #blue put: [:fontDesc | fontDesc color:ColorValue blue].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@400.
composedText text emphasizeAllWith: #red.
composedText displayOn: gc at: 400@450.
composedText text emphasizeAllWith: #blue.
composedText displayOn: gc at: 400@500
```

Example 2: Defining font size emphasis – a problem requiring a two-argument character attribute block

Problem: Define a new emphasis symbol #size used in size -> 20, such that a character assigned this emphasis is displayed with pixel size specified as its argument.

Solution: This problem requires a character attribute block with two argument, the second of which will be the size. The size argument will be used to assign pixel size. The general procedure is the same as in Example 1.

```
| composedText ca ta gc |
composedText:= 'A string' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #size put: [:fontDesc :size| fontDesc pixelSize: (fontDesc pixelSize + size)].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@400.
composedText text emphasizeAllWith: #size -> 16.
composedText displayOn: gc at: 400@450.
composedText text emphasizeAllWith: #size -> 20.
composedText displayOn: gc at: 400@500
```

Example 3: Display a part of the text using a different font family.

Problem: We are to display the string 'This string uses a different family' using font AvantGarde as indicated. The rest is displayed using default font and using the indicted styles.

Solution: We will use the same procedure as in the previous examples with the default font and a new text style called #avantgard. The program is as follows:

```
| composedText ca ta gc |
composedText:= 'This string uses a different family' asComposedText.
"Construct CharacterAttributes."
ca := CharacterAttributes newWithDefaultAttributes.
ca setDefaultQuery: text textStyle defaultFont.
ca at: #avantgarde put: [:fontDesc | fontDesc family: 'avantgarde*'].
ta := TextAttributes characterAttributes: ca.
```

```
composedText textStyle: ta.  
text text emphasizeFrom: 20 to: 36 with: #avantgarde.  
gc := Window currentWindow graphicsContext.  
composedText displayOn: gc at: 400@450.
```

Note that we used the wild card character in 'avantgarde*' because we were not sure of the exact name of the font family.

Example 4: Display the *whole* text using a non-default font family and combine it with various other styles.

Problem: We want to display the whole string 'This string uses a different font family'. using AvantGarde again and emphasize the whole text by underlining as shown.

Solution: In this case, it is better to modify the base font from which the whole string is defined. This is done by creating a new FontDescription object and using it as the default font for the CharacterAttributes. The program is as follows:

```
| composedText ca ta gc fontDescription |  
composedText:= 'This string uses a different font family' asComposedText.  
"Create a new base font description object."  
fontDescription := FontDescription new  
family: 'avantgarde*';  
pixelSize: 16.  
"Construct CharacterAttributes."  
ca := CharacterAttributes newWithDefaultAttributes.  
ca setDefaultQuery: fontDescription.  
ca at: #avantgarde put: [:fontDesc | fontDesc family: 'avantgarde*'].  
ta := TextAttributes characterAttributes: ca.  
composedText textStyle: ta.  
composedText text emphasizeAllWith: #underline.  
gc := Window currentWindow graphicsContext.  
composedText displayOn: gc at: 400@450
```

Example 5: Find fonts available on the current platform.

Problem: We may want to allow the user to select any font from a multiple choice dialog (Figure A.8.4). How do we find which fonts are available?

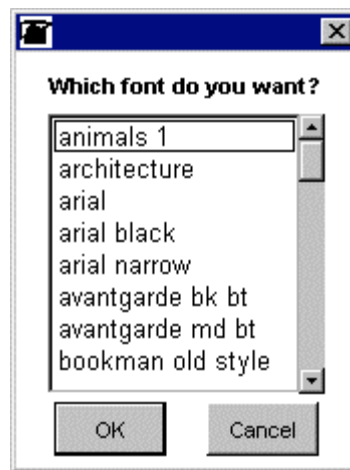


Figure A.8.4. Multiple dialog offering a choice from all available fonts.

Solution: The 'graphic device' on which we are printing (in this case the screen) knows about available fonts. A possible solution is as follows:

```
| fontFamilies fontFamily |
"Collect all available font families."
fontFamilies := (Screen default defaultFontPolicy availableFonts collect:
                [:fontDesc | fontDesc family]) asSet asSortedCollection.
"Allow user to select a family."
fontFamily := Dialog choose: 'Which font do you want?'
                fromList: fontFamilies
                values: fontFamilies
                lines: 8
                cancel: [#noChoice]
```

Example 6: Allow the user to select a style and display the stylized text in the current window.

Problem: Use the multiple choice window from the previous example to choose a font family from all families available on the current platform, and display text using this family.

Solution: The principle of the solution is to use the collection of font families to add new emphases to the dictionary. The program is as follows:

```
| fontFamilies fontFamily ca composedText ta gc |
composedText := 'Experimental text.' asComposedText.
"Get available font families."
fontFamilies := (Screen default defaultFontPolicy availableFonts
                collect: [:fontDesc | fontDesc family]) asSet asSortedCollection.
"Construct CharacterAttributes."
(ca := CharacterAttributes newWithDefaultAttributes)
    setDefaultQuery: composedText textStyle defaultFont.
"Create dictionary with all font styles."
fontFamilies do: [:family | ca at: family asSymbol put: [:fontDesc | fontDesc family: family]].
"Let user select a font family."
fontFamily := Dialog choose: 'Which font do you want?'
                fromList: fontFamilies
                values: fontFamilies
                lines: 8
                cancel: [#noChoice].
ta := TextAttributes characterAttributes: ca.
composedText textStyle: ta;
    text emphasizeAllWith: fontFamily asSymbol.
gc := Window currentWindow graphicsContext.
composedText displayOn: gc at: 400@500
```

Example 7: Controlling font in a text editor widget

Problem: Implement a window with a text editor widget and an <operate> menu with command *font* that allows the user to select any platform font for the currently highlighted text. The *font* command should open a multiple choice window as in the previous example, from which the font selection is made.

Solution: We will reuse the techniques from previous examples. The only new aspect of this problem is how to force the new emphasis on the selection in the text editor widget and this task is performed simply by changing the emphasis of the current text editor selection, assigning it as the new selection, and invalidating the widget. The methods implementing the application are as follows:

Initialization

initialize

"Define context of the text editor widget."

```
text := Array comment asComposedText asValue
```

postBuildWith: aBuilder

```
"Create new TextAttributes with all platform fonts."  
| ca ta |  
fontFamilies := (Screen default defaultFontPolicy availableFonts  
                  collect: [:fontDesc | fontDesc family]) asSet asSortedCollection.  
ca := CharacterAttributes newWithDefaultAttributes.  
ca setDefaultQuery: text value textStyle defaultFont.  
fontFamilies do: [:family | ca at: family asSymbol put: [:fontDesc | fontDesc family: family]].  
ta := TextAttributes characterAttributes: ca.  
widget := (self builder componentAt: #textWidget) widget.  
widget textStyle: ta
```

Menu

menu

```
"Extend built-in text editor menu."  
| mb |  
mb := MenuBuilder new.  
mb addDefaultTextMenu;  
    line;  
    add: 'font family' -> #fontFamily;  
    add: 'font size' -> #fontSize.  
^mb menu
```

where command *font family* is implemented by

fontFamily

```
"Let user select a font family and redisplay the selection with the new font."  
| fontFamily selection |  
(selection := widget controller selection) isEmpty ifTrue: [^self].  
(fontFamily := Dialog choose: 'Which font family do you want?'  
                           fromList: fontFamilies  
                           values: fontFamilies  
                           lines: 8  
                           cancel: [nil]) isNil ifTrue: [^self].  
self applyNewEmphasis: fontFamily asSymbol onText: selection
```

which uses

selectFontFamily

```
"Let user select a font family."  
^Dialog choose: 'Which font do you want?'  
              fromList: fontFamilies  
              values: fontFamilies  
              lines: 8  
              cancel: [#noChoice]
```

and command *font size* is implemented by

fontSize

```
"Let user select a font family and redisplay the selection with the new font."  
| fontSize selection |  
(selection := widget controller selection) isEmpty ifTrue: [^self].
```

```
(fontSize := Dialog choose: 'Which font size do you want?'
fromList: #(#small #normal #large)
values: #(#small #normal #large)
lines: 3
cancel: [nil]) isNil ifTrue: [^self].
self applyNewEmphasis: fontSize onText: selection.
widget invalidate
```

Finally, the method that applies emphasis and refreshes the widget is

applyNewEmphasis: aSymbol onText: aText

```
"Emphasize current selection with chosen emphasis."
aText emphasizeAllWith: aSymbol.
widget controller replaceSelectionWith: aText.
widget invalidate
```

Closing notes

FontPolicy

You might be wondering how **FontPolicy** which converts the desired font to an available font does this work. The principle is that if it does not find the desired font, it takes the parameters of the desired font (an instance of **FontDescription**) and compares them with the parameters of available fonts, and finds one that matches the desired font most closely. In doing this, it applies various weighting coefficients to all parameters as explained in the class comment whose essential part is as follows:

I represent a policy mapping **FontDescriptions** to actual **ImplementationFonts** available on a particular device. Mapping is done by assigning weights to the various properties that a **FontDescription** can have, and using those weights to assign a quality value to each of the available fonts as it is compared to a font request. A quality value of 0 indicates an exact match. A high quality value indicates a poor match. Any concrete font whose quality is greater than the policy's tolerance is removed from consideration.

Assigning weights can be somewhat tricky. Perhaps the easiest approach is to start by choosing a default tolerance (for example, the system default is 9), and then choose the other weightings relative to that. For example, the default system assumes that the size of characters and whether they are fixed width is moderately important, so they have weights of 3. Boldness is a little less important, and serifness even less so. The name of a font is very specific, so it's given a very high weight of 10, because the user probably has very high expectations of seeing a particular font.

Note that you can be easily redefine the weighting attributes or the whole scheme of font matching if you wish to do so. In fact, you may want to redefine the whole font management architecture if you find it unsatisfactory, for example for some of the reason mentioned in the following notes.

Notes on TextAttributes

Our presentation implies that each **ComposedText** has exactly one **TextAttributes**. Since this object defines properties that apply not only to individual characters but also to the text as a whole, the whole object is displayed with the same alignment (centered, left aligned, etc.), the same line separation, and so on. If this is not desirable, the text must be divided into sections of **ComposedText** elements with their own parameters.

Note also that the default value of **TextAttributes** is shared by all **ComposedText** objects that use it. As a consequence, changing its parameters changes text attributes for all **ComposedText** objects

that use it. When defining new text styles, a new instance of **TextAttributes** should thus be created – as we have been doing all along. `textattr. shared – make copy`

Finally, a note on emphases and the attributes dictionary. If an undefined emphasis (one that does not appear as a key in the attributes dictionary) is encountered during display, this emphasis is ignored. If a sequence of conflicting emphasis is encountered (such as a sequence of emphases specifying different font sizes), the last one is applied.

Exercises

1. We mentioned that the fact that the whole **ComposedText** object shares the same **TextAttributes** may be too limiting. Since text editor widgets are designed to use a single **ComposedText** object as their model, this imposes limits on the extendibility of text editor widgets. The easiest way to obtain a window with sufficient text processing power thus seems to be to use subviews and appropriately extended existing views supporting **ComposedText** display - the **ComposedTextView** view, and the **ParagraphEditor** controller. Use this approach to create a window with a subview that allows the user to control the font family, pixel size, color, and other text parameters via <operate> menu commands.
2. Modify the implementation from Exercise 1 to implement font control using an interface similar to Microsoft window font controls.
3. Extend the above exercises to allow simple drawing. This exercise is better suited for a project.
4. Extend the previous exercise to allow 'pluggable' drawing tools selectable from a menu. This exercise is also better suited for a project.

A.8.3. Drag-and-Drop

Drag and Drop is the familiar process of selecting an item in a source widget by pressing the <select> mouse button over an item, dragging the mouse pointer over another widget with the button pressed, and dropping the data into a target widget by releasing the mouse button over it. The action is usually accompanied by a visual feedback, typically by changing the shape of the mouse pointer as it moves over windows and widgets. In VisualWorks, the source may be a list, and the target may be a window or any widget.

We will give a simple example of the implementation of drag and drop but first the principles. The operation involves the cooperation of the following new classes:

- **DragDropManager**. An instance of this class coordinates the whole drag and drop operation from the moment the user presses the <select> button over a source widget, to the moment the button is released over a target.
- **DragDropContext** is carried by **DragDropManager** and contains all information necessary for the operation including the dragged data (an instance of **DragDropData**), information about cursor shape, and information stored in it by a drop target, usually the one that was most recently entered. An instance of this class is used as the argument of messages sent by **DragDropManager** as it carries out the drag and drop operation and provides access to all necessary drag and drop information.
- **DragDropData** holds data to be transferred and information about the drag origin widget. The data is often held in an **IdentityDictionary** which makes it possible to store any number of items and access them by arbitrary keys. It also contains a key object, a **Symbol** identifying the nature of the data for use by the target.
- **DropSource** is used to provide information about cursor shapes at various stages during the operation.
- **ConfigurableDropTarget** is an object representing the target widget. It is automatically created by the **UIBuilder** when a widget's properties specifies that the widget is a drag and drop target. When the mouse pointer moves into this widget's bounds, the **DragAndDropManager** recognizes this and sends mouse motion-related messages to it.

The basic operation of drag and drop is as follows (Figure A.8.5):

1. User presses <select> button over a widget identified as a source via its **DragStart** property.
2. The source widget sends its **DragStart** message to the application.
3. The **DragStart** method is responsible for creating a **DragDropData** instance containing the data and possibly other information such as a **Symbol** identifying the nature of the data. The method also creates a **DropSource** object containing information about entry effects. Finally, the **DragStart** method must create an instance of **DragDropManager** which will be responsible for the whole drag and drop. The **DragDropManager** now takes over.
4. As the user moves the mouse with the pressed button, the **DragDropManager** monitors the windows and widgets over which the mouse pointer is passing. If the mouse passes over a window or widget set up as a drop target via its properties, it sends the following messages to the application model:
 - a designated *entry* message (a specified property of the widget) – when it enters a target widget
 - a designated *over* message (a specified property of the widget) – when the mouse pointer moves while over a target widget
 - a designated *exit* message (a specified property of the widget) – when it exits a target widgetAll these messages have a **DragContext** as their argument and obtain information such as the data and the key **Symbol** from it. They typically provide visual feedback by changing the cursor or highlighting the widget.
5. When the user releases the button over a target window or widget, **DragDropManager** sends a designated **Drop** message (a specified property of the widget) to the application. This message typically processes the data and provides visual feedback.
6. The drag and drop operation is finished and the **DragDropManager** is released.

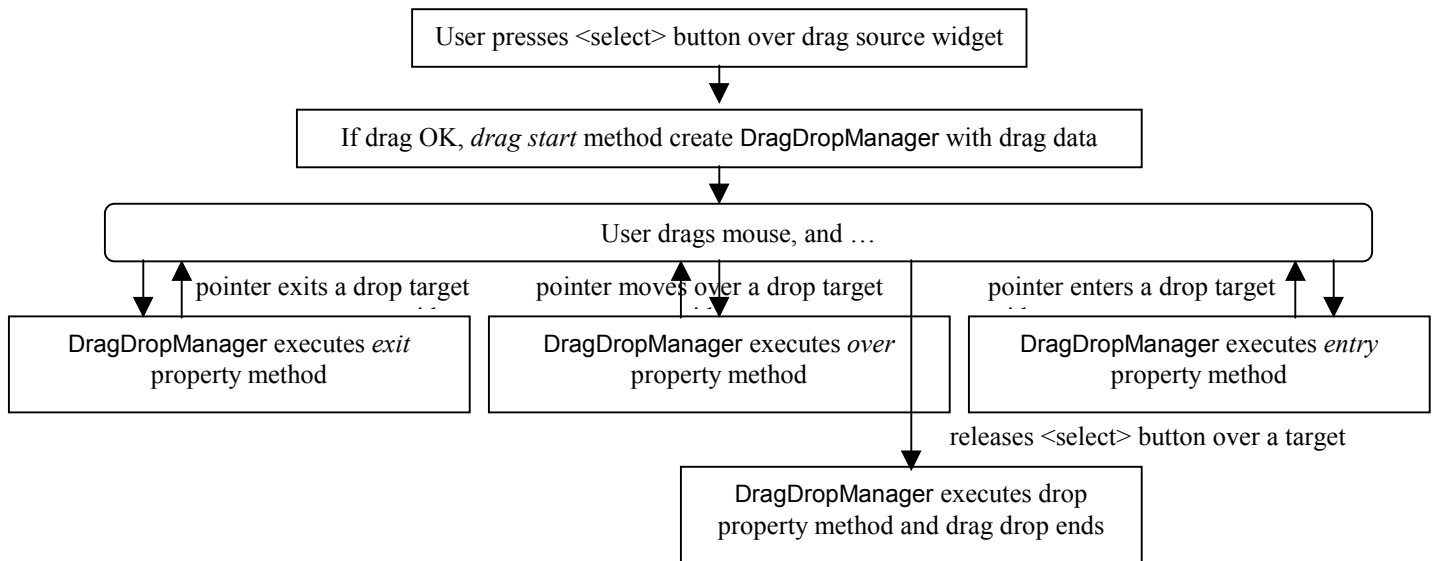


Figure A.8.5. Essence of the Drag and Drop process.

After this introduction, we will now present a simple example of a drag and drop user interface.

Example: Library Cataloguing Tool

Problem: Assume that a part of the process of cataloguing new books is classifying them as 'regular' books and 'reserve' books. Assume that all new books are entered into a list, and a 'reserve clerk' examines each book and classifies it manually as regular or reserve. Our task is to implement a preliminary version of a tool to computerize this task (Figure A.8.6). To use this tool, the clerk first clicks a book title in the list which displays its information, and then drags the title to one of the list below. At this point, the book is removed from the original list.

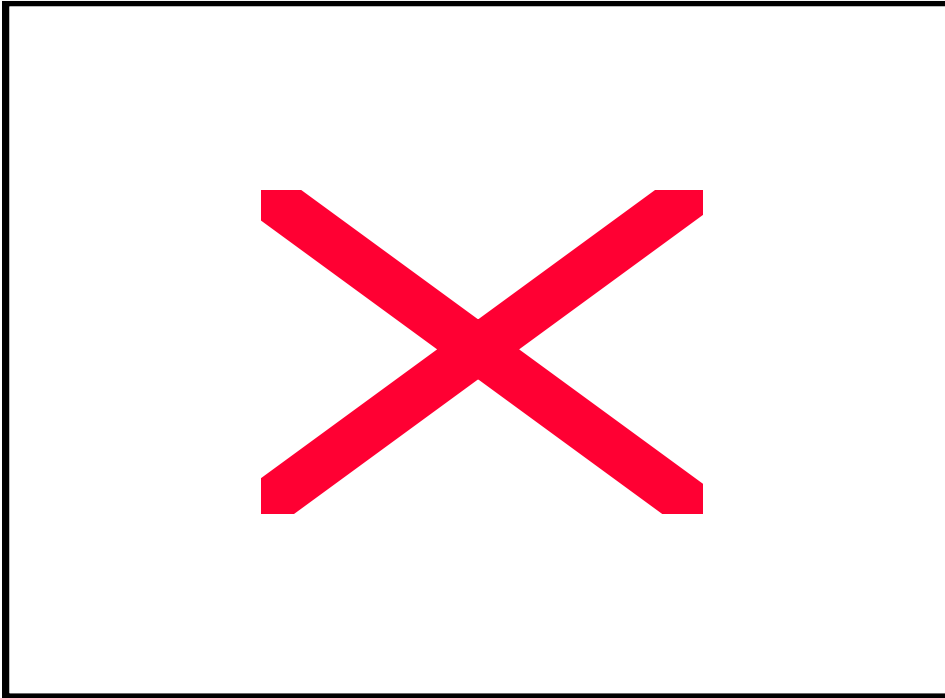


Figure A.8.6. Book classification tool.

Design

Assume that we already have a suitable class `Book` to represent individual books. To keep things simple for this example, we will hold the new books and the classified books in the application model's list aspect variables `newBooks`, `regularBooks`, and `reserveBooks`. The application model class will be called `BookClassifier`.

For the sake of this example, we will assume that the new books list already exists (we will initialize it to a few books during initialization) and the only responsibility of `BookClassifier` will be the drag and drop operation. In addition to the collection instance variables, `BookClassifier` will have aspect variables `author`, `title` and `year` for the input fields.

Implementation

Draw the user interface We specified the following properties for the three lists:

New Books – drop source:

Drag OK: `#dragOK`

Drag Start: `#dragStart`

Notification:

Change: `#newSelection` - equivalent to sending `onChangeSend: newSelection to: self` on initialization

Regular Books – drop target:

Entry: `#entry`

Over: `#over`

Exit: `#exit`

Drop: `#addRegularBook`

Reserve Books – drop target:

Entry: `#entry`

Over: #over
Exit: #exit
Drop: #addReserveBook

Note that we specified the same Entry, Over, and Exit methods for both drop target lists because we want the same visualization behavior for both: Both will change the cursor in the same way on entry and on exit.

We will now implement initialization and all the above methods.

initialize

“Create test list of new books and assign it to the New Books list widget.”

```
| books authors titles years |
books := SortedCollection sortBlock: [:x :y | x author < y author].
authors := #('Orwell' 'Steinbeck' 'Wright' 'Moliere' 'Carre' 'Ross' 'Turgenev' 'Crane' 'Richardson').
titles := #('Nineteen Eighty-Four' 'Of Mice and Man' 'Black Boy' 'Tartuffe' 'The Little Drummer Girl'
            'As For Me and My House' 'Fathers and Sons' 'The Red Badge of Courage' 'Wacousta').
years := #(1980 1975 1960 1670 1992 1980 1985 1970 1977).
1 to: authors size do: [:index | books add: (Book
    author: (authors at: index)
    title: (titles at: index)
    year: (years at: index))].

self newBooks list: books
```

Now the drag and drop related methods, starting with the drag source widget. The dragOK method is executed when the user presses <select> over the widget. It returns true if drag is OK, false otherwise.

dragOK: aController

“Allow drag operation only if the list is not empty.”
^newBooks list isEmpty not

If dragOK returns true, the following method sets up a DragDropManager and starts the drag operation. All the messages in the method are required for drag and drop to work.

dragStart: aController

“Create and strta dd with data necessary for visualization and data drop.”
| data ds dm |
data := DragDropData new.
data clientData: newBooks selection.
data contextWindow: self builder window.
data contextWidget: aController view.
data contextApplication: self.
ds := DropSource new.
dm := DragDropManager withDropSource: ds withData: data.
dm doDragDrop

Proceeding now to the target lists, we will first define the entry, over, and exit methods that respond to the passing of the mouse into, over, and out of the widget. They all return a Symbol which is used by DropSource to select the appropriate cursor. This value – an *effect symbol* – has the following default values: #dropEffectMove, #dropEffectNone, #dropEffectCopy, and #dropEffectNormal, associated with special cursors; custom cursors may be defined for user-defined symbols as well. Our definitions don’t do anything except for returning the appropriate effect symbol:

entry: aDragContext

“Show that it is possible to move the object to this list.”
^#dropEffectMove

over: aDragContext

"Show that it is possible to move the object to this list."
^#dropEffectMove

exit: aDragContext

"Show that no drop operation is possible outside the list."
^#dropEffectNone

Finally, we will define the drop method when the user executes the drop operation by releasing the mouse button. The method returns the effect symbol defining the shape of the cursor after the operation. For the Regular Books list the method is

addRegularBook: aDragContext

"User released mouse button. Get data and add it to this list, remove the item from the New Books list."
| book |
book := aDragContext sourceData clientData.
regularBooks list: ((regularBooks list) add: book; yourself).
self removeBook: book.
^#dropEffectNone

and addReserveBook: is similar. Both methods share removeBook: which is defined as follows:

removeBook: aBook

"User dropped aBook into another book list, remove it from the New Books list."
newBooks list: ((newBooks list) remove: aBook; yourself).
title value: ".
author value: ".
year value: "

This completes the implementation and the program is now fully functional. As mentioned in the introduction, this example shows only the basic features of Drag and Drop and we encourage you to study the more involved examples in the Cookbook, and to implement the exercises.

Main Lessons Learned

- Drag and drop involves widgets whose properties define them as drop source or drop targets.
- A widget may be a drop source, a drop target, both, or none of these.
- Drop source properties include the name of a method that determines whether it is OK to perform a drag and drop, and a method that creates a **DragDropManager** with appropriate data to realize the operation.
- Drop target properties include the name of methods that determine what happens when the mouse pointer enters the widget, moves inside it, and leaves it. They all return an effect symbol which determines the new shape of the mouse cursor. Another method is activated when the user releases the mouse; it executes the drop action and returns an effect symbol.
- The Notification property of a widget may be used with the same effect as **onChangeSend:to:** during initialization.

Exercises

1. Implement the example from the text.
2. We
3. Extend the example by allowing the user to drag a book from Regular Books to Reserve Books and vice versa. This is possible because a widget may be both a source and a target.
4. Extend the example by allowing the user to drag a book from Regular Books or Reserve Books to a garbage bin. (A label may be an image and a drop widget.)
5. Add a list of Archived Books reserved for old books. Only a book published before 1800 may be dropped into Archived Books. (Hint: Calculate a Symbol - #old or #notOld – when a book is picked, assign it as key to the **DragDropContext**, and use it to make a decision.)

A.8.4 The Virtual Machine

Every programming language that allows its users to create objects dynamically must have a mechanism for removing objects that are no longer needed. Otherwise, many applications would soon run out of memory. According to the mechanism for destroying unneeded objects, programming languages can be divided into two groups: those that destroy unneeded objects automatically (such as Smalltalk, Java, and LISP), and those that require the programmer to destroy unneeded objects by explicit destructor construction (such as C++). Modern programming practices generally prefer automatic garbage collection.

Automatic garbage collection is several decades old and underwent a lot of evolution because its inefficient implementation may render it practically unusable. In terms of strategies, there are two basic approaches to identifying inactive objects (also known as ‘corpses’): One is to associate a count of existing references for every new object and increment or decrement the count when a reference to the object is created or dropped. The other approach is to establish a part of the system as the ‘root’ and decide whether an object is live by attempting to trace a chain of references from the roots of the system to the object. Both strategies are recursive in that the marking of referenced objects requires proceeding down to object components until finding a primitive object with no components, and that incrementing or decrementing the count also requires going down to the primitive objects. The disadvantage of reference counting is that it must be done whenever an object is created, destroyed, or assigned and this approach is thus inefficient and not used any more.

In addition to an algorithm for distinguishing live objects from corpses, we also need a storage strategy and a strategy for deciding whether all objects need to be examined or not, and if all objects are not examined on every garbage collection pass then which ones are and which ones are not. As the experience with garbage collection grew, it was discovered that a very large majority of objects created during execution have a very short life span while other objects are very stable. This means that once an object remains live long enough, it will probably remain so for a very long time and testing it is a waste of time. On the basis of this finding, modern garbage collection techniques divide objects into several categories, store them in separate memory spaces, and deal with them separately.

After this general introduction, we will now describe how garbage collection is performed in VisualWorks. Our description is based on comments and other information available in class `ObjectMemory` which is responsible for performing garbage collection, and class `MemoryPolicy` which defines the garbage collection strategy and parameters.

Garbage collection in VisualWorks

A.8.5 Garbage collection

Conclusion

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

`ConfigurableDropTarget`, *DispatchTable*, `DragDropContext`, **`DragDropData`**, **`DragDropManager`**, **`DropSource`**, *LookPreferences*, **`ParagraphEditor`**,