5

ANIMATING THE PROGRAM

A SECOND CLASS

For *which of you, desiring, to build a tower, does not first sit down and count the cost, whether he has enough to complete it?*
THE HOLY BIBLE, Revised Standard Version

If you have worked in the computing industry and have tried to convince management to follow your ideas, you know that while a written proposal is nice, a good demo really sells the project. One of the nice things about Smalltalk is that a good demo is easy to create. In this chapter we will improve our program by adding animation, which produces excellent demos. There are three kinds of animation commonly used in Smalltalk, and we will tackle the simplest kind first. Our program will use black rectangles to represent the disks, and they will travel in two dimensions on a white background (see Figure 5.1). In this first version, the movement of the disks will be jumpy, not smooth. In an exercise in Appendix 4, we will meet class OpaqueForm and use the follow:while: message to produce smooth movement. Dan Ingalls

Figure 5.1

wrote a more sophisticated animation package, but it is not uniformly available to Smalltalk users, so we will not cover this third kind of animation here.

In a conventional programming language, a single large program would manage the position on the screen of each of the disks. It would call a routine to display images of the disks. The Smalltalk style is to make each disk be an object, that is, a package of data and procedures that belong together. A disk object is responsible for keeping track of its position on the screen and for displaying itself. We want the object that represents the whole game—the single instance of TowerOf-Hanoi—to know almost nothing about animation. Thus we divided the Tower of Hanoi game into objects, and we are about to make an object-oriented version of the program. These objects are really rather familiar, and they are pictured in Figure 5.2.

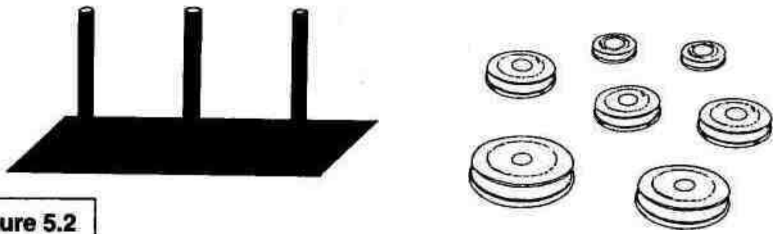


Figure 5.2

The definition for the class that represents the disks looks like this:

```
Object subclass: #HanoiDisk
  instanceVariableNames: 'name width pole rectangle '
  classVariableNames: TheTowers Thickness DiskGap'
  poolDictionaries:''
  category: 'Kernel-Objects'
```

The definition of class HanoiDisk says that every disk owns four variables. Three additional variables are shared across the whole class. Their values can be read or written by any instance of the class. The comment in class HanoiDisk explains what the variables are for:

```
Each disk in the game is represented by an object of class HanoiDisk.
It has
   name-name of this disk (a Character)
   width-size of the disk (1 is the smallest disk width)
   pole-number telling which pole the disk is on
   rectangle-a rectangle on the screen that the disk occupies

There are three variables shared across the whole class
   TheTowers-the object that represents the whole game
      and holds the stacks of disks
   Thickness-the thickness of a disk in screen dots
   DiskGap-the number of screen dots between disks in a stack
```

We have just mentioned a distinction between two kinds of variables. An individual instance of class HanoiDisk represents one disk. It has its own distinct values for the "instance variables" called name, width, pole, and rectangle. For the entire class, there is just one value of each of the "class variables" called TheTowers, Thickness, and DiskGap. The single value of a class variable is shared by all instances of the class. Variables that are shared across the whole class are capitalized.

Local variables are another kind of variable we've discussed, and they are easy to understand. A local variable comes into being when a method begins execution, and goes away when the method terminates. Its name is meaningless outside of that individual method. The major difference between local and instance variables is that instance variables live as long as the objects of which they are a part, usually much longer than the execution of a single method. (Class variables live as long as the class is in existence, usually forever.)

As long as we are discussing variables, let's cover one last kind. A global variable is available anywhere in the system, and its name always begins with a capital letter. We have already seen many globals without knowing it. The way we access the object that represents a particular class, such as Array, is to use the global variable of the same name. In addition, there are a few globals whose values are not classes, such as Transcript.

Just to firm up the distinction between instance variables and class variables, let's look at the Pascal code that roughly corresponds to the definition of class HanoiDisk. (Keep in mind that we are really stretching things to make this analogy—don't get into any arguments with friends based on this.)

```
Program Hanoi
  Var TheTowers: TowerOfHanoi;
    Thickness:     Integer;
    DiskCap:       Integer;
  Record HanoiDisk =
    name:   Character;
    width:  Integer;
    pole:   Integer;
    rectangle:     Rectangle;
    end.
```

Before we discuss the methods in class HanoiDisk, let's look at the changes to TowerOfHanoi. Although TowerOfHanoi knows nothing about the mechanics of the animation, it still needs a few changes. For example, it must create instances of HanoiDisk and push them onto the stacks instead of using characters. We could make changes directly to class TowerOfHanoi, but then we would not be able to run the old non-animated version of the game. Instead, let's define a new class that is a "subclass" of TowerOfHanoi. A subclass inherits all the definitions of variables and methods of the parent class. The subclass can add new variables, add new methods, and override old methods by redefining them. The class AnimatedTowerOfHanoi is declared as a subclass of TowerOfHanoi as follows:

```
TowerOfHanoi subclass: #AnimatedTowerOfHanoi
  instanceVariableNaines: 'howMany  mockDisks '
  dassVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Objects'
```

The comment for this class is:

```
An object of this class represents the game. It inherits the variable stacks
from class TowerOfHanoi. The new instance variables are:
  howMany-the number of disks
  mockDisks-an Array of fake disks (when a disk asks what disk
    it can move on top of, and the pole is empty, we return a mock disk;
    it has nearly infinite width)
```

The simplest way to explain the code in our two new classes is to consider first a move in the middle of the game. After we have looked at all the code in both classes for an average move, we will examine the initialization code. The actual moving of disks takes place during moveDisk:to:. Let's write a new version in class AnimatedTowerOfHanoi that overrides the one in TowerOfHanoi. Changes from the older version are underlined.

```
moveDisk: fromPin to: toPin
   | disk supportDisk |
   supportDisk <- (stacks at: toPin) isEmpty
      ifFalse: [(stacks at: toPin) first]
      ifTrue: [mockDisks at: toPin].
   disk <- (stacks at: fromPin) removeFirst.
   (stacks at: toPin) addFirst: disk.
   "inform the disk and show move"
   disk moveUpon: supportDisk.
^   Transcript cr.
   Transcript show: (fromPin printString,' -> ', toPin printString,'').
   Transcript nextPut: disk name.
   Transcript endEntry."_
```

Starting in the middle of the method, the disk that is being moved (disk) is now an instance of HanoiDisk. When it is sent the message moveUpon:, it moves its image on the screen. The way it determines where to place itself is by looking at the disk it is moving on top of, supportDisk. When moved, a disk centers itself above a lower, supporting disk. What if the pole the disk is moving to is empty? The easiest way to handle this exceptional condition is to supply a fake disk as the supportDisk. The fake disk contains the pole number to move to and a position on the screen. The variable mockDisks is an array of three fake disks at the bases of the three poles. The new code at the beginning of the method

```
   supportDisk <- (stacks at: toPin) isEmpty
      ifFalse: [(stacks at: toPin) first]
      ifTrue: [mockDisks at: toPin].
```

assigns the proper mock disk to supportDisk if the stack is empty. If the stack has disks, the top disk is assigned to supportDisk.

We just used the message ifFalse:ifTrue:. The if-then-else control message comes in four flavors. The message ifTrue: executes the statements in the block if it was sent to the object true. If false was the receiver, control passes to the next statement in the program. Likewise, ifFalse: executes the block when sent to false. The message ifTrue:ifFalse: executes just one of its two blocks, depending on the conditional expression it was sent to. Finally, ifFalse:ifTrue: does exactly the same thing, but is simply written with the other block first. (Both forms are provided so that you can write whichever half you think of first.)

At the bottom of the method moveDisk:to: we modified the printing code to show the name of the disk (disk used to be just a character).

We will define the method name in HanoiDisk to allow each disk to return the character that is its name. We enclosed the entire printing code in comment quotes so that printing in the transcript window will not interrupt the animation. (For debugging, the comment quotes can be removed.)

The method moveUpon: is defined in class HanoiDisk and looks like this:

```
moveUpon: destination
   "This disk just moved. Record the new pole and tell the user."
   pole <- destination pole.
   "remove the old image"
   self invert.
   "reposition"
   rectangle center: destination center - (0 @ (Thickness + DiskGap)).
   "display the new one"
   self invert.
   (Delay forMilliseconds: 300) wait
```

In the assignment statement, we set this disk's new pole to be the destination disk's pole. The message pole, sent to the destination disk, asks what pole it is on. Next, we send ourselves the message invert to "exclusive or" our rectangle onto the screen.* This removes our image at the disk's old location. In the next line, we assign our rectangle a new center point. Then we invert again to place an image on the screen at the new location, and then pause for 300 milliseconds. Since pole and rectangle are instance variables, they retain their new values until we decide to move this disk again.

Before we plunge into the code for positioning disks, we need to discuss the coordinate system and the objects that stand for points and rectangles. The point (0,0) is at the upper left corner of the screen. The X axis goes across and the *positive* Y coordinates go down the screen, implying an upside-down coordinate system. The arithmetic operator @ makes a point out of two integers. Thus the expression 100@300 returns a point (an instance of class Point) whose X value is 100 and whose Y value is 300. This point is 100 screen dots out from the left edge and 300 dots down. Strictly speaking, in the expression 100@300, object 100 is an instance of SmallInteger (an integer) and it understands the message @, which has one argument. The @ method creates a new Point, and assigns it X and Y values. Points understand most arithmetic operations, such as +, -,*,//, =, and abs. (// means divide and truncate to an integer, abs means take the absolute value.)

* The operation "exclusive or" on two rectangles combines their contents to produce a third. It puts black where the bits in the two rectangles are different, and white where they are the same. 1 bits represent black and 0 bits are white, so the bitwise operation is exclusive or.

Let's examine the code for repositioning the rectangle.

```
rectangle center: destination center - (0 @ (Thickness + DiskGap)).
```

A Rectangle is a bundle of four points that are the comers of a rectangle. We will compute our rectangle's new center by subtracting a quantity from the destination disk's center. (Subtracting a positive quantity means moving up on the screen.) Thickness + DiskGap is the spacing between disk centers. 0 @ (Thickness + DiskGap) creates a point whose X is 0 and whose Y value is the height of a disk. Subtraction between points is "vector subtraction," and yields a new point whose Y is the difference between the two points' Y values. The message center: moves our disk's rectangle to a new location without changing its size. Note that there are two very similar message names, center and center:. They have the same name except for the colon. There is an informal convention among Smalltalk programmers that pairs of messages like center and center: are related. The one without the colon asks for a value and the one with the colon sets the value.

Definitions of the classes Point and Rectangle are found in the System Browser in the category **Graphics-Primitives,** which is the first **Graphics-** category. The blue Smalltalk book explains the full story of Points and Rectangles in Chapter 18.

We still have some messages for class HanoiDisk that have not been defined. These are used in moveUpon:.

```
pole
   "return which pole this disk is on"
   t pole
```

Every Smalltalk method that runs returns a single object as its result. If you don't specify anything to be returned, the object that received the message (self) is returned. To return some other object, put an up-arrow, \ , in front of the expression for the object you want to return. In this case, the whole purpose of the method is to return this disk's pole number, so that another disk can compare its pole number with it.

```
center
   "return a Point that is the current center of this disk"
   f rectangle center
```

Note that center (without the colon) is already a message in class Rectangle. Any rectangle that receives it computes and returns its center point.

Alas, the message center: (with the colon) is not defined as "standard equipment" in class Rectangle. One virtue of Smalltalk is that the user can correct oversights in the basic system. When we type in the methods for class HanoiDisk into the system, we will have to correct this oversight. We will make an excursion up to class Rectangle and define the message center: as

```
center: thePoint      | extent |
   "move the rectangle so it is centered on the point,
      but keep the width and height unchanged"
   extent«- corner - origin.
   origin <- thePoint - (extent // 2).
   corner <— origin + extent
```

An instance of class Rectangle only stores two points internally: the upper left and the lower right point. It computes the remaining points ifthey are needed. The message center: takes a single argument and stores it in thePoint. The goal is to move the rectangle so that its new center point is thePoint. We save the size of the rectangle in the local variable extent. The new upper left-hand corner (origin) is the center minus one-half the extent. The new lower right-hand corner (corner) is the origin plus the extent. (Systems programming was never so easy.)

The code for moveUpon: sends another message that we have not defined. Here is the code to "exclusive or" the image ofa disk onto the screen:

```
invert
   "show a disk on the screen by turning white to black in a
      rectangular region "
   Display reverse: rectangle
```

Now let's look at the initialization code for the animated version ofour game. A new version ofhanoi in class AnimatedTowerOfHanoi will override the one in TowerOf Hanoi.

```
hanoi      | aString |
   "Ask the user how many disks, set up the game, and move disks until
      we are done."
   aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
   howMany «- aString asNumber.
   self setUpDisks.      "create the disks and stacks"

   self moveTower: howMany from: 1 to: 3 using: 2.

   " (AnimatedTowerOfHanoi new) hanoi "
```

The first difference is that the number of disks in the game is now stored in howMany instead of height, height was a temporary variable (indicated by its appearing between vertical bars at the beginning of the method). Its value was not accessible to methods that were called several levels below hanoi. We have made howMany an instance variable so that any disk can ask for it later (see below). Second, we have moved all of the initialization of the stacks of disks into a new subroutine called setUpDisks.

```
setUpDisks      | disk displayBox |
  "Create the disks and set up the poles."
  "Tell all disks what game they are in and set disk thickness and gap"
  HanoiDisk new whichTowers: self.
  displayBox <- 20@100 corner: 380@320.
  Display white: displayBox.
  Display border: displayBox width: 2.
  "The poles are an array of three stacks. Each stack is an
    OrderedCollection."
  stacks <- (Array new: 3) collect: [:each | OrderedCollection new].
  howMany to: 1 by: -1 do: [:size |
    disk <- HanoiDisk new width: size pole: 1. "Create a disk"
    (stacks at: 1) addFirst: disk.      "Push it onto a stack"
    disk invert "show on the screen"].

  "When a pole has no disk on it, one of these mock disks acts as a bottom
    disk. A moving disk will ask a mock disk its width and pole number"
  mockDisks <- Array new: 3.
  1 to: 3 do: [:index |
    mockDisks at: index put: (HanoiDisk new width; 1000 pole: index)].
```

Undaunted by the size of this method, let's examine its parts.

```
  HanoiDisk new whichTowers: self.
```

Class HanoiDisk has three variables that are shared class-wide: TheTowers, Thickness, and DiskGap. Later, we will define the message whichTowers: to initialize them. It only has to be sent once, and this seems like a good place to do it.

The next statement creates a white area with a black border on the screen in which we will put the images of our disks.

```
  displayBox <- 20@100 corner: 380@320.
  Display white: displayBox.
  Display border: displayBox width: 2.
```

We specify the area to make white by creating a rectangle. The message corner: sent to a Point causes it to create a new instance of class Rectangle. In this case, its upper left point (20,100) receives the message and takes the lower right point (380,320) as an argument.

Display is a global variable that holds the object that represents the screen. It is an instance of DisplayScreen and inherits much of its behavior from the basic classes that represent bit images; Form, DisplayMedium, and DisplayObject. The BitBit operation actually moves images around. Chapters 18, 19, and 20 of the Blue Book explain these classes, BitBit, and displaying in general.

The message white: is one member of a family of rectangle-filling methods. The others are black:, gray:, lightGray:, darkQray:, and veryLightGray:. We place a black border two dots wide inside the white rectangle by sending the message border:width: to Display. To display more complicated images, one generally creates an instance of class Form, makes an image inside of it, and transfers the image in the Form to the display. A Form can be treated like a brush, and moved along a path, such as a line or a curve.

The next section of code hasn't changed much:

```
stacks <- (Array new: 3) collect: [:each | OrderedCollection new].
howMany to: 1 by: -1 do: [:size [
   disk <- HanoiDisk new width: size pole: 1. "Create a disk"
   (stacks at: 1) addFirst: disk.      "Push it onto a stack"
   disk invert "show on the screen"].
```

stacks is an array of three stacks. We fill the first stack in reverse order with instances of HanoiDisk. Each new instance of HanoiDisk is immediately sent the message width:pole: to set its width and which pole it is on. Inside width:pole: the disk's name and rectangle are set. In the last line we tell the disk to display itself.

We are ready for the final part of setUpDisks.

```
mockDisks <— Array new: 3.
1 to: 3 do: [:index |
   mockDisks at: index put: (HanoiDisk new width: 1000 pole: index)].
```

The mock disks are the fake disks used by moveDisk:to: to represent poles that do not have any disks on them. There is one for each pole and it is located just below where the first disk would go. We give the mock disks a width of 1000 to make them so wide that any disk can move on top of them, using the normal rules of the game. The message at:put: stores a value in a particular place in an Array. The first argument is the index within the Array and the second argument is the value to be stored. The message at: retrieves the value from the location in the Array specified by the argument.

At the beginning of setUpDisks, the statement HanoiDisk new whichTowers: self initialized the class variables in HanoiDisk. Here is the code:

```
whichTowers: aTowerOfHanoi
   "Install the object representing the towers"
   TheTowers <- aTowerOfHanoi.
   Thickness <— 14.      "thickness of a disk in screen dots"
   DiskGap <— 2.      "distance between disks"
```

It is clear why all disks need to know a thickness and a distance from other disks on a stack, but the purpose of TheTowers is less clear. Communication is crucial between AnimatedTowerOfHanoi and the disks. They communicate by sending messages to each other. If the main program wants to ask a disk something like, "How wide are you?", it asks the appropriate disk object on one of the stacks. If, on the other hand, a disk trying to position itself on the screen needs to ask the object that represents the whole game a question, what does it call that object? The variable TheTowers holds the instance of TowerOfHanoi. A disk can say TheTowers howMany to ask the instance of Animated-TowerOfHanoi how many disks there are total. At the beginning of setUpDisks in the statement (HanoiDisk new whichTowers: self), self is the object representing the whole game. That object is stored into aTowerOfHanoi and then into TheTowers.

Now let's examine the initialization code for a HanoiDisk.

```
width: size pole: whichPole      | where y |
   "set the values for this disk"
   width <- size.
   pote <- whichPole.
   "compute the center of the disk on the screen"
   size >= 1000 ifFalse: ["a normal disk"
         name <- Character value: ($A asciiValue) + size -1.
         y<-300 - ((TheTowers howMany - size)*(Thickness+DiskGap)).
         where <—100@y]
      ifTrue: ["a mock disk"
         name <- $m.
         where <- (100*whichPole) @ (300 + Thickness + DiskGap)].
   "create the rectangle, specify its size, and locate its center"
   rectangle «- 0@0 extent: (size*14)@Thickness.
   rectangle center: where.
```

First we set the disk's own width and pole. We are actually creating two different kinds of disks. Mock disks will not show on the screen, but real disks will use the locations ofthe centers of mock disks to stack themselves in the right places. Regular disks have both a width and a position on the Hrst pole that depend on their size. The message "greater than or equal to" is written as >=. A normal disk gets a computed value for its name, and a mock disk gets the character *m*.

Next we compute the local variable where to hold the position of the disk. We'll put the three stacks at X positions of 100, 200, and 300. The bottoms ofall stacks will be at 300 in Y. N*(Thickness + DiskGap) is the height of N disks. Inside class HanoiDisk, we don't know the total number of disks. To find that out, we must ask TheTower, because it holds all of the game-wide information. (TheTowers howMany — size)*(Thickness + DiskGap) is the distance above the base of the stack of a disk with width size. The bottom disk is the widest and the top disk is the narrowest (with size = 1). Given the size, these two statements create a point, where, for the center of a normal disk.

```
y<-300 - ((TheTowers howMany - size)*(Thickness + DiskGap)).
where *-100@y.
```

If a disk's size >= 1000, then the disk is a mock disk. Its position is below the start of a stack, and the stack for each pole is in a different place. Here is a mock disk's location:

```
where <- (100*whichPole) @ (300 + Thickness + DiskGap).
```

The statement

```
rectangle<-0@0 extent: (size* 14) ©Thickness.
```

creates a Rectangle at (0,0) with the proper width and height. The final statement

```
rectangle center: where.
```

moves the Rectangle so that its center is at the point we computed.

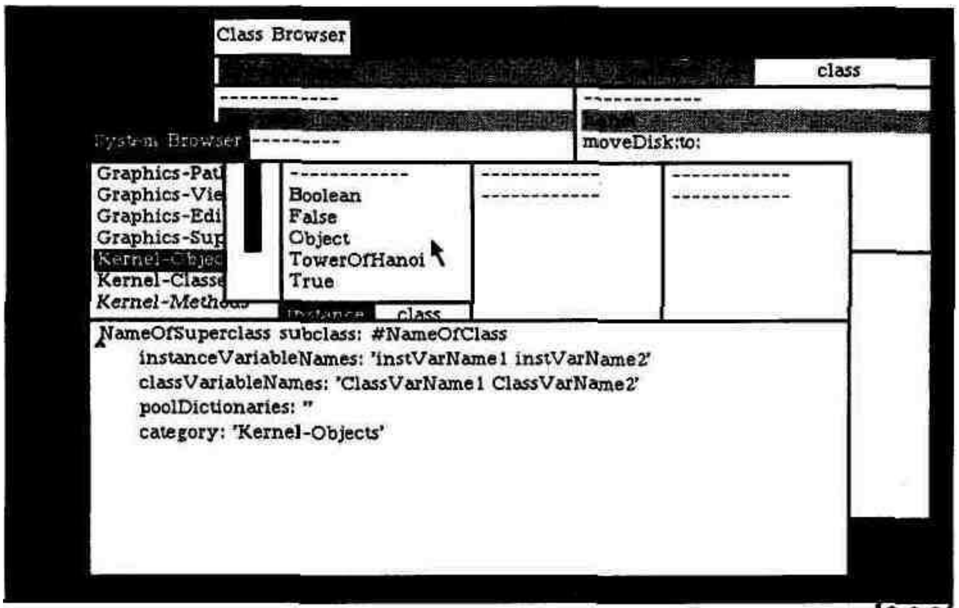
We have called the method howMany in AnimatedTowerOfHanoi, but we have neglected to define it. The code simply returns the value we saved in an instance variable.

```
howMany
"return the number of disks"
 f howMany
```

We've seen a lot of new things in this chapter: class variables, objects of two different classes interacting, and a class that is a subclass of another one. All the code we looked at may seem a bit confusing now, but it will become more coherent after you get it into the browser.

INSTALLING THE CLASSES HanoiDisk AND AnimatedTowerOfHanoi

Your screen probably has two browsers on it, a Class Browser for TowerOfHanoi and the original System Browser. Let's create class HanoiDisk first and do it in the System Browser. Enter the System Browser and deselect anything that is selected in area B (see Figure 5.3).



```
NameOfSuperclass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Kernel-Objects'
```

Figure 5.3

Modify the template in area E to be:

```
Object subclass: # HanoiDisk
  instanceVariableNames: 'name width pole rectangle '
  classVariableNames: TheTowers Thickness DiskGap '
  poolDictionaries: ''
  category: 'Kernel-Objects'
```

accept and move to area B. Choose **comment** from the middle-button menu and install this class comment. (Once again, if your systern is a License 1 system, be sure to put the comment inside the single quotes in area E.)

```
Each disk in the game is represented by an object of class HanoiDisk.
It has
   name-name of this disk (a Character)
   width-size of the disk (1 is the smallest disk width)
   pole-number telling which pole the disk is on
   rectangle-a rectangle on the screen that the disk occupies

There are three variables shared across the whole class
   TheTowers-the object that represents the whole game
      and holds the stacks of disks
   Thickness-the thickness of a disk in screen dots
   DiskGap-the number of screen dots between disks in a stack
```

accept, then choose **protocols** from the middle-button menu in area B. We will divide the messages in class HanoiDisk into two groups called **access** and **moving.** Type this in area E and **accept.**

```
('access')
('moving')
```

Let's enter the methods in the **access** category. In area C, select **access** and type and **accept** each of these methods individually in area E.

```
pole
   "return which pole this disk is on"
   f pole
```

The character ^ is usually on the A key (shift 6).

```
name
   "return the name of this disk"
   f name
```

```
whichTowers: aTowerOfHanoi
   "Install the object representing the towers"
   TheTowers *- aTowerOfHanoi.
   Thickness <-14.   "thickness of a disk in screen dots"
   DiskGap <- 2.     "distance between disks"
```

Anytime that you want to see the class definition again (to remind yourself what the instance variables are named), you can choose **defi-**

nition from the middle-button menu in area B. Back in the **access** category, define the method for width:pole: as follows:

```
width: size pole: whichPole     | where y |
   "set the values for this disk"
   width <- size.
   pole <- whichPole.
   "compute the center of the disk on the screen"
   size >= 1000 ifFalse: ["a normal disk"
         name <- Character value: ($A asciiValue) + size -1.
         y«-300 - ((TheTowers howMany - size)*(Thickness+DiskGap)).
         where <— 100@y]
      ifTrue: ["a mock disk"
         name <- $m.
         where <- (100*whichPole) @ (300 + Thickness + DiskGap)].
   "create the rectangle, specify its size, and locate its center"
   rectangle <- 0@0 extent: (size*14)@Thickness.
   rectangle center: where.
```

When you **accept,** the compiler may put up a menu to ask you if howMany is a new method name. If so, click on **proceed** as is (or on the item **howMany,** if it appears)*. Now let's continue by replacing the contents of area E with the following method:

```
center
   "return a Point that is the current center of this disk"
   f rectangle center
```

Now we need to go to class Rectangle and add the message center:. After accepting the message you just typed in area E, go up to area A. Scroll up the list of categories in area A until you see **Graphics-Primitives.** It is the uppermost category of the **Graphics-** type. Select it, then select Rectangle in area B and **accessing** in area C. In area E replace the template with:

```
center: thePoint   | extent |
   "move the rectangle so it is centered on the point,
     but keep the width and height unchanged"
   extent <- corner - origin.
   origin *- thePoint - (extent // 2).
   corner <- origin + extent
```

* In some systems, the compiler will also ask you to conErm center: as a new message.

accept it and then scroll back in area A to Kernel-Objects. Select it, then select HanoiDisk, then select moving. Now we'll fill in the methods that have to do with moving disks.

```
moveUpon: destination
   "This disk just moved. Record the new pole and tell the user."
   pole <- destination pole.
   "remove the old image"
   self invert.
   "reposition"
   rectangle center: destination center - (0 @ (Thickness + DiskGap)).
   "display the new one"
   self invert.
   (Delay forMilliseconds: 300) wait
```

The compiler will fret because invert is an unfamiliar message name. Gently reassure it by clicking on **proceed as is** (choose the name of the method if you are using a License 1 system from Apple).

```
invert
   "show a disk on the screen by turning white to black in a
      rectangular region "
   Display reverse: rectangle
```

Let's create class AnimatedTowerOfHanoi in the System Browser and copy methods from the Class Browser for TowerOfHanoi. Enter the System Browser and deselect anything that is selected in area B.

Edit the template in area E until it looks like this:

```
TowerOfHanoi subclass: #AnimatedTowerOfHanoi
   instance VariableNames: 'howMany mockDisks '
   classVariableNames:'_
   poolDictionaries: ''
   category: 'Kernel-Objects'
```

After you've accepted that, set the comment to be:

```
An object of this class represents the game. It inherits the variable stacks
from class TowerOfHanoi. The new instance variables are
   howMany-the number of disks
   mockDisks-an Array of fake disks (when a disk asks what disk
      it can move on top of, and the pole is empty, we return a mock disk;
      it has nearly infinite width)
```

accept and move up to area B. Choose **protocols** from the middle-button menu and replace the contents of area E with

```
('the game')
```

and **accept.** In area C, be sure that **the game** is selected. Then type and **accept** each of these methods individually in area E. (If you wish, go to the other browser and copy the hanoi method and then modify it.)

```
hanoi     [ aString |
   "Ask the user how many disks, set up the game, and move disks until
      we are done."
   aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
   howMany <- aString asNumber.
   self setUpDisks.      "create the disks and stacks"

   self moveTower: howMany from: 1 to: 3 using: 2.

   " (AnimatedTowerOfHanoi new) hanoi "
```

The compiler will become suspicious of the message setUpDisks. Tell it to stop being so stuffy by choosing **proceed as is** (choose the name of the method if you are using a License 1 system from Apple). Now type and **accept** the method:

```
setUpDisks      | disk displayBox |
   "Create the disks and set up the poles."
   "Tell all disks what game they are in and set disk thickness and gap"
   HanoiDisk new whichTowers: self.
   displayBox <- 20@100 comer: 380@320.
   Display white: displayBox.
   Display border: displayBox width: 2.
   "The poles are an array of three stacks. Each stack is an
      OrderedCollection."
   stacks <— (Array new: 3) collect: [:each | OrderedCollection new].
   howMany to: 1 by: -1 do: [:size |
      disk <- HanoiDisk new width: size pole: 1.      "Create a disk"
      (stacks at: 1) addFirst: disk.      "Push it onto a stack"
      disk invert "show on the screen"].

   "When a pole has no disk on it, one of these mock disks acts as a bottom
      disk. A moving disk will ask a mock disk its width and pole number"
   mockDisks <— Array new: 3.
   1 to: 3 do: [:index |
      mockDisks at: index put: (HanoiDisk new width: 1000 pole: index)].
```

Go to the Class Browser and copy the moveDisk.'to: method, bring it into this category, and then modify it as shown by the underlining below:

```
moveDisk: fromPin to: toPin
  [ disk supportDisk [
  supportDisk <- (stacks at: toPin) isEmpty
     IfFalse: [(stacks at: toPin) first]
     ifTrue: [mockDisks at: toPin].
  disk <- (stacks at: fromPin) removeFirst.
  (stacks at: toPin) addFirst: disk.
  "inform the disk and show move"
  disk moveUpon: supportDisk.
^  Transcript cr.
  Transcript show: (fromPin printString,' - > ', toPin printString,'').
  Transcript nextPut: disk name.
  Transcript endEntry. "
```

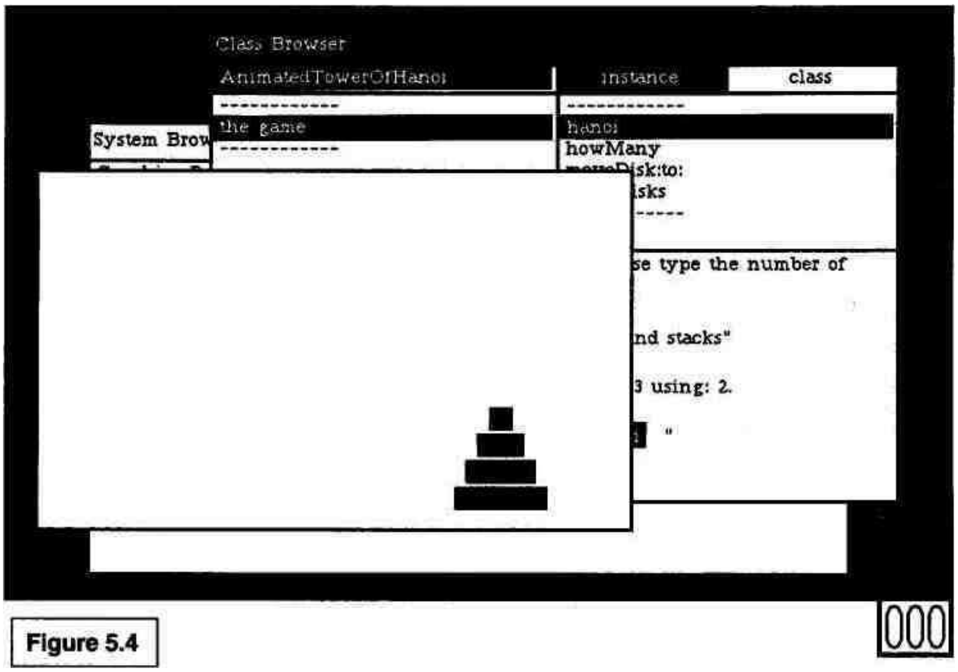
```
howMany
  "return the number of disks"
  f howMany
```

We are finished entering AnimatedTowerOf Hanoi. Let's replace the Class Browser for TowerOfHanoi with one for AnimatedTowerOfHanoi. Enter the Class Browser, and choose **close** from the right-button menu. Back in the System Browser, go to area B and choose **spawn (browse** in License 1 systems) from the middle-button menu. Position the corner cursors to frame your new browser.

Now let's see some moving disks! Remember to accept the method you just typed. Enter the Class Browser for AnimatedTowerOfHanoi, go to the hanoi method, and scroll to the comment at the bottom of the code. Select (AnimatedTowerOfHanoi new) hanoi, and choose **do it.** See the section "Troubleshooting Runtime Errors" in Chapter 3 if the program has bugs (see Figure 5.4).

Notice that the white area in which the animation is running is not really a window. Our program wrote on the screen directly, without enclosing the drawing in a window that can become active after it has been covered up. In order to keep this example simple, we will not discuss how to create a new kind of window. (Later, when you have finished this book, you can examine the classes in the category **Interface-File Model** and use them as an example.)

You now have some experience with graphics in Smalltalk. One of



```
se type the number of

nd stacks"

3 using: 2.
```

Figure 5.4

the exercises in Appendix 4 explores a smoother and fancier kind of animation.

The class AnimatedTowerOfHanoi is a subclass ofTowerOfHanoi, from which it inherits instance variables and methods. In this chapter, we have used inheritance (also called subclassing) in all three of the ways it is used in Smalltalk. First, we used it to add to the behavior of a class, TowerOfHanoi, by adding instance variables, overriding existing methods, and adding new methods. Second, we used subclassing as a protection mechanism. While the animation code was "in pieces on the floor" being written and debugged, our previous example in TowerOfHanoi still worked. All of the changes and new code were isolated in the subclass, leaving the original class untouched. The third use of inheritance was illustrated when we created class TowerOfHanoi in the previous chapter. Making a subclass is the only way to make a new class in Smalltalk. The greatest independence a new class can have is to be a subclass of class Object. In that case, all it inherits is "objectness," that is, the property of being an object in the Smalltalk system.