4

DEFINING A CLASS

THE CLASS TowerOfHanoi

If thought corrupts language,
language can also corrupt thought.
GEORGE ORWELL, *Nineteen Eighty-Four*

We bet several skeptical readers have doubts about whether this Tower of Hanoi algorithm really works. The disks are never identified or stored anywhere. The recursive calls leapfrog, one on top of another, and we really can't tell by looking at the code whether the algorithm works. If you were asked to print which disks were on which stacks after each move, you could not do it with the present data structures. We are now ready to use honest data structures and convince the skeptics.

We are going to make a data structure of three stacks and move the disks between them. Rather than have a separate variable for each stack, an array will hold the stacks so we can index them by pole number. The name of the variable for the array of stacks is stacks. We will use the characters A, B, C, etc., to represent the disks. What kind of variable should stacks be? In a conventional language, a local variable in the outer procedure would hold the towers. But Smalltalk uses a different, object-oriented style; it allows you to create a new kind of object to represent the Tower of Hanoi game—an object which contains variables to hold the stacks of disks.

In Smalltalk, you describe a new type of object before creating it. When you are done, the description works just as well for a whole class of objects. We call such an object description a "class." Any object created from the description is called an "instance" of the class. After we define a class of objects that holds the state of the Tower of Hanoi game, we have the power to create many examples, or instances, of the game with different numbers of disks and in different stages of progress through the game.

The definition of a class always begins by filling in a template.

```
Object subclass: #TowerOf Hanoi
  instanceVariableNames: 'stacks'
  classVariableNames:''
  poolDictionaries:''
  category: 'Kernel-Objects'
```

In the next section we will see how to enter this definition into the system.

Every class in the Smalltalk system should have a comment associated with it. The comment is used to tell what the variables mean, and to explain what the class is for. The comment below for the class TowerOfHanoi is short, because we know you will have to type it later.

```
stacks Is an Array of stacks.
    Each stack is an OrderedColtection.
    The objects we put on the stacks are characters.
    A is the smallest disk, B is larger, etc.
    addFirst: is the message for push, and removeFirst is pop.
```

For two of the three methods we have written, we are going to make a new version of the code inside class TowerOfHanoi. Here is the new version ofhanoi.

```
hanoi
  "Tower of Hanoi program. Asks user for height of stack of disks"

  | height aString |
  aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
  height <- aString asNumber.
  Transcript cr.
  Transcript show: Tower of Hanoi for:', height printString.
  stacks <- (Array new: 3) collect: [:each | OrderedColtection new],
  (height to: 1 by: -1) do: [:each | (stacks at: 1) addFirst:
      (Character value: ($A asciiValue) + each -1)].
  self moveTower: height from: 1 to: 3 using: 2.

        (TowerOfHanoi new) hanoi.
```

This method serves to illustrate a few more features of the Smalltalk language. Let's walk through the code, commenting on the new language features as we find them. The first few lines are unchanged from before, hanoi has two local variables and asks the user for the number of disks. We want stacks to be a 3-element array, with each element of the array being a stack of disks. Since there are no type declarations in Smalltalk, we need to execute statements to create objects of the right type (instances of the right class). The statement assigning a value to Stacks does this, and we will examine it in pieces. Smalltalk uses the new: method to create a new object. To make an array of 3 elements, you just say

```
Array new: 3
```

and you make a stack by saying

```
OrderedCollection new
```

(Smalltalk generally calls aggregate data types "collections"; and since the elements in a stack are ordered, a stack is considered an "ordered collection.")

We are going to skip the sixth line in the method hanoi, and consider the seventh and eighth lines. When we go back and explain the sixth, it will be easier to understand. The Smalltalk statement on the seventh and eighth lines is a "do loop." Besides iteration, it also includes array indexing, number conversion, and stack operations. We'll explain the pieces before we show you the whole thing. The standard Smalltalk iteration message is do:, and a fine example is

```
(1 to: height) do: [:each | statements].
```

This does exactly what you would expect, with each being the iteration variable whose value goes from 1 to height by increments of 1. The colon in :each means that a new value of each will be supplied each time the block is evaluated (as usual with iteration variables). The vertical bar acknowledges the fact that we are creating each as a local variable to the block (delimited by brackets). It can only be used within the block. A backwards counting loop would be

```
(height to: 1 by: -1) do: [:each | statements}.
```

How do you index arrays? The at: message selector (operator) is the answer, so you refer to the first element of stacks by

```
stacks at: 1.
```

The remaining operations create characters that represent disks and then push them onto a stack. Using standard methods that are part of the Smalltalk system, you can convert an integer n to its corresponding ASCII character by

```
Character value: n.
```

The method to push an element onto a stack is addFirst:.

Now put it all together. Disk A should be the top disk, so push the last disk onto the stack last. From height to 1 by minus 1, make the corresponding character (. . . C, B, A) and push it onto the stack of tower number 1:

```
(height to: 1 by: -1) do: [:each | (stacks at: 1) addFirst:
  (Character value: ($A asciiValue) + each -1)].
```

The second line takes the ASCII value of the character A and adds the disk number minus 1. It then turns that back into a character to give the letters of the alphabet in order. $A is the literal character A.

Looking again at the sixth line of the hanoi method, where the stacks are created:

```
stacks <- (Array new: 3) collect: [:each | OrderedCollection new].
```

Smalltalk has several methods for transforming aggregate data types (collections); collect: is a popular example. The general form is

```
x collect: y.
```

This creates a new collection the same size as x, performing the operations specified in the block of code y to initialize each element. Let's make an array with 3 elements, with each element a *new* stack (OrderedCollection), and store the result in stacks:

```
stacks «- (Array new: 3) collect: [:each | OrderedCollection new].
```

Think of collect: as being just like do:. The local variable each is like an iteration variable, except that its values come from the collection (Array new: 3). Notice that after the value of each is assigned (on each iteration), we are totally ignoring it. The new values that we are collecting, a bunch of brand-new OrderedCollection's, can be constructed without reference to what was in the array before. (You may

be curious what value each actually has. It is, in turn, each of the nils in the array we just created.) The message collect: is, in fact, a control structure that both creates a new collection and runs a loop to get its initial values.

As before, the last line of the hanoi method (before the comment) sends the method moveTower:from:to:using: to self.

Now let's look at the new method for moveDisk:to:.

```
moveDisk: fromPin to: toPin
  "Move a disk from a pin to another pin. Print the results in the
    transcript window"
  | disk |
  disk <- (stacks at: fromPin) removeFirst.
  (stacks at: toPin) addFirst: disk.
  Transcript cr.
  Transcript show: (fromPin printString,' -> ', toPin printString,'').
  Transcript nextPut: disk.
  Transcript endEntry
```

The method pops a disk off the "from" stack into the local variable disk and pushes it onto the "to" stack. Then it prints the move along with the name of the disk that was moved. You already know how to index (at:) and push an element (addFirst:). You might guess that removeFirst pops the stack, so you pop from one and push to another by

```
disk <- (stacks at: fromPin) removeFirst.
(stacks at: toPin) addFirst: disk.
```

In the System Transcript window we print a carriage return, the "from" pole, the "to" pole, and append a space. The message nextPut: sends a single character (the disk's name) to the transcript. We must send endEntry afterwards to make the window refresh itself and show the character. (We didn't do this in the previous version because the method for show: has its own call on endEntry.)

HOW TO CREATE A NEW CLASS

Picking up our narrative, you'll remember we left you just after you heroically figured out how to save the universe by stopping the Tower of Hanoi with 64 disks. Now that the universe is assured of continuing, we can create a new class called TowerOfHanoi. In area A of the browser, make sure that the category **Kernel-Object** is selected.

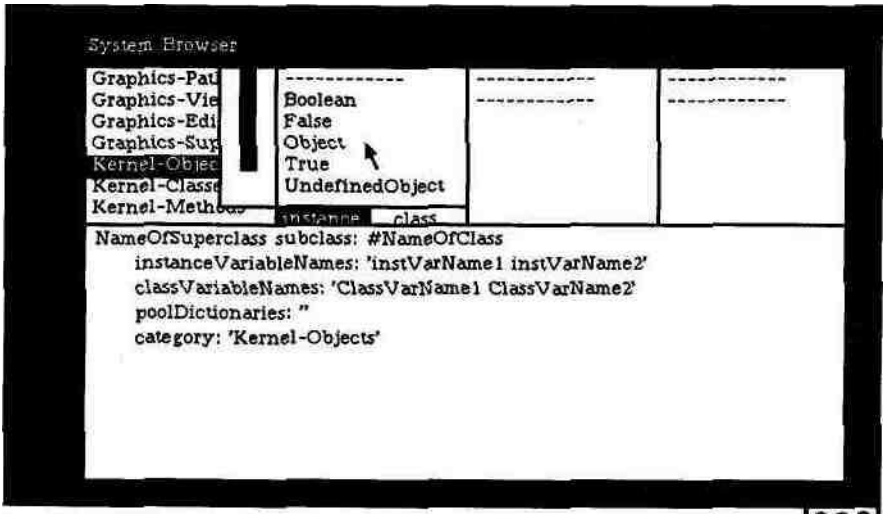

Figure 4.1

In area B, deselect the class Object (click on it) if it is currently selected (see Figure 4.1).

Replace the template in area E with the definition below. If you don't want to type the whole thing, you can edit the template in area E until it looks exactly like the definition below. The parts you need to change are underlined. (Be sure to **cut** out the stuff inside the single quotes at the end of the third line.) When you are checking the code you have typed against the code here, you may find it helpful to move the cursor along to keep your place on the screen.

```
Object subclass: #TowerOfHanoi
   instanceVariableNames: 'stacks'
   classVariableNames: '_
   poolDictionaries:''
   category: 'Kernel-Objects'
```

accept the definition (middle-button pop-up menu in area E). Sections 12.1 to 12.4 in the User's Guide have more details about class definitions. TowerOfHanoi should appear in the list in area B.

Now that you know about classes, the layout of the browser window should make more sense. Area A is a menu of categories of classes. Area B is a menu of individual classes within the category you have chosen. Area C is a menu of categories of methods (these are called "protocols") within the currently selected class. In area D, individual method names appear in the menu. In particular, moving across our

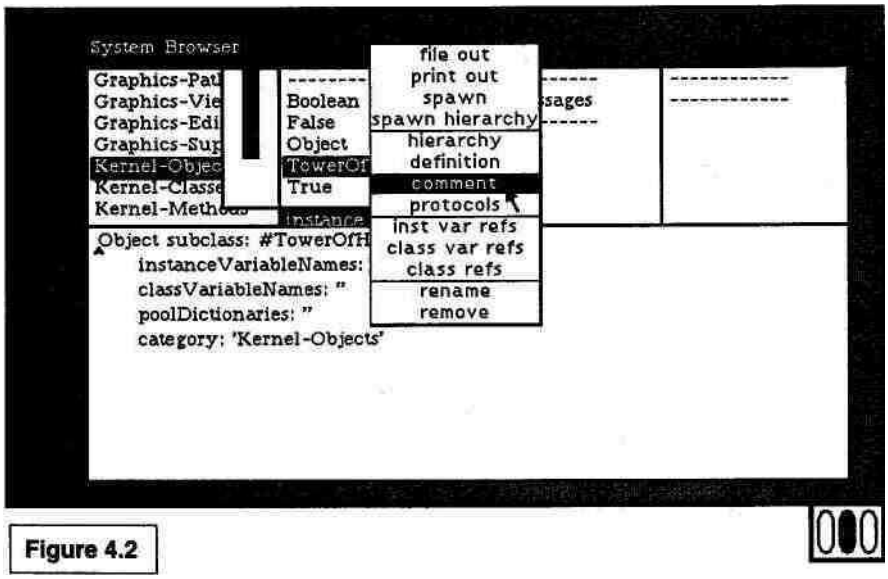

Figure 4.2

browser, we are in the category **Kernel-Objects** and are looking at the class TowerOf Hanoi. We are going to define the message category **games** within the class and then install the method hanoi. The middle-button pop-up menu in each area has actions and requests that operate on the thing you have selected in that menu.

Every class should have a comment. In area B, choose **comment** from the middle-button menu (see Figure 4.2).

Now type the explanatory comment below. (License 1 users beware. You must type your comment inside the single quotes in area E. Do not erase the text that says TowerOfHanoi comment:. All normal License 2 users can simply replace the entire contents of area E with the text below.)

```
stacks is an Array of stacks.
    Each stack is an OrderedCollection.
    The objects we put on the stacks are characters.
    A is the smallest disk, B is larger, and so on.
    addFirst: is the message for push, and removeFirst is pop.
```

Say **accept,** move to area C (the message category menu), and add the category **the game.** We've added categories before; use the middle-button pop-up menu to choose **add protocol,** type *the game,* and press the return key. (If you are a License 1 user, you must go to area B and choose **protocols** from the middle-button menu. Type *('the game')* in area E and **accept.)** An entry for **the game** should appear in the message category menu, area C (License 1 users should enter area C and select it).

Now move the methods we already have into the new class you created. Rather than making you type them again, Smalltalk lets you use the text editor to move them from one place in the Smalltalk class structure to another. The easiest way to do this is to create a second browser that is dedicated to this new class and then use the **copy** and **paste** commands to move the methods from the old browser to the new one:

(1) To create a new browser, enter area B (the class names menu) and choose **spawn** from the middle-button pop-up menu (it's called **browse** in License 1 systems).

(2) Smalltalk will ask you where you want to place the new browser on the display. The question will be phrased as a corner cursor (see Figure 4.3). Imagine where you want your new browser

Figure 4.3 

On the screen and what shape it should be. Try to pick an area that overlaps as little as possible the area of the old browser. Move the cursor to the best place for the upper left-hand corner of the new browser. Once there, hold the left button down and the cursor will change to a lower right corner cursor (see Figure 4.4).

Figure 4.4 

While still holding the button, move the mouse to where you want the lower right hand corner of the browser. When you let up on the button you will see the new browser. This operation with the corner cursor is called "framing a window." You can reframe any Smalltalk window at any time by using the right-button menu and selecting **frame.**

(3) The new browser (Figure 4.5) looks different because it is for only one class, namely TowerOfHanoi. This browser is really the same as the old one, except that area A is gone, and the others are rearranged. Figure 4.6 identifies the areas of the new browser, which is called a Class Browser.

(4) Go back to the System Browser (the original browser). Wake it up by pointing in it and clicking. This brings it to the front of the display, onto the top of the stack of overlapping windows (see Figure 4.7).

(5) Select the class Object in area B (the class name menu) of the System Browser and the category **games** in area C (you may

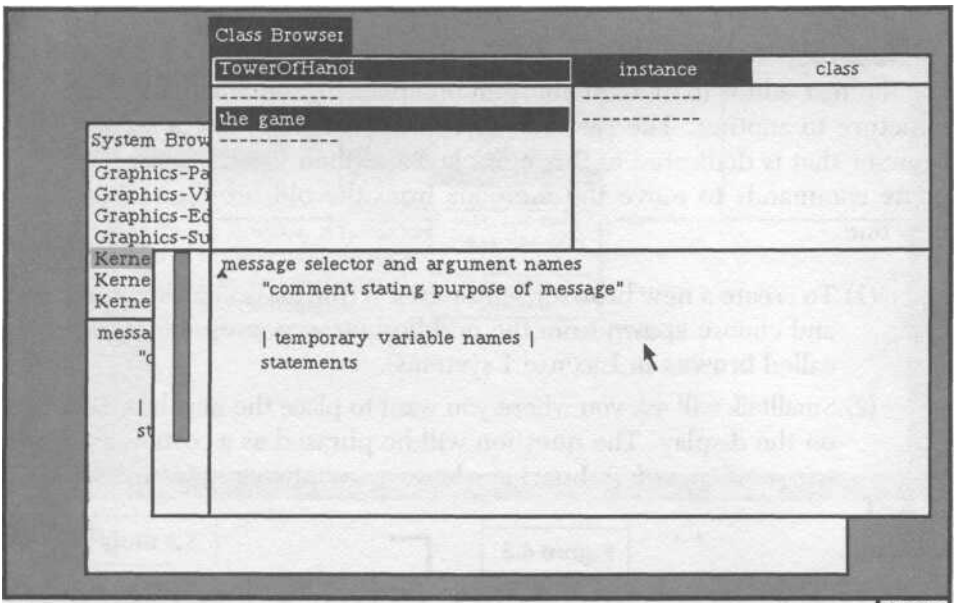

Figure 4.5

have to scroll to find it; use the upward-pointing arrow). Select hanoi in area D.

(6) Go down to the text area E of the System Browser and select all the text (reverse-video everything). (If you can't see the

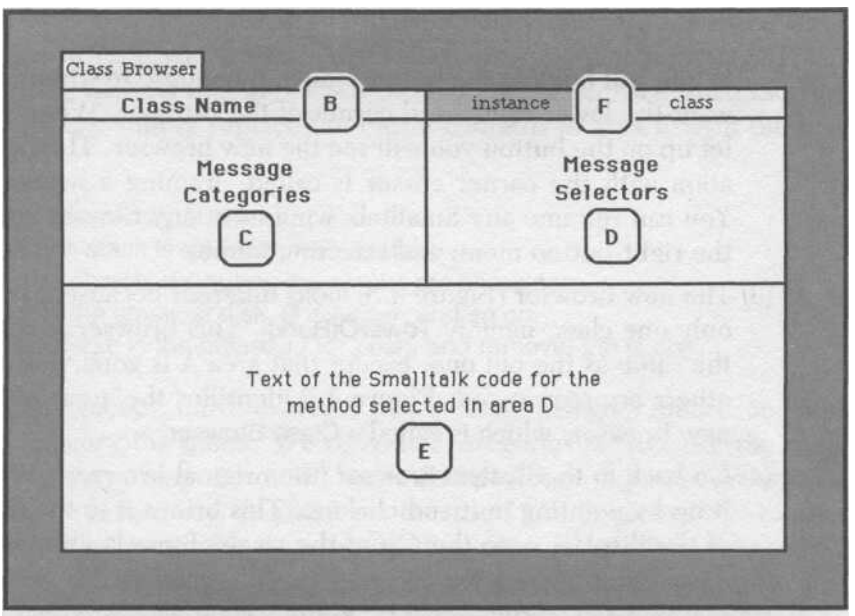

Figure 4.6

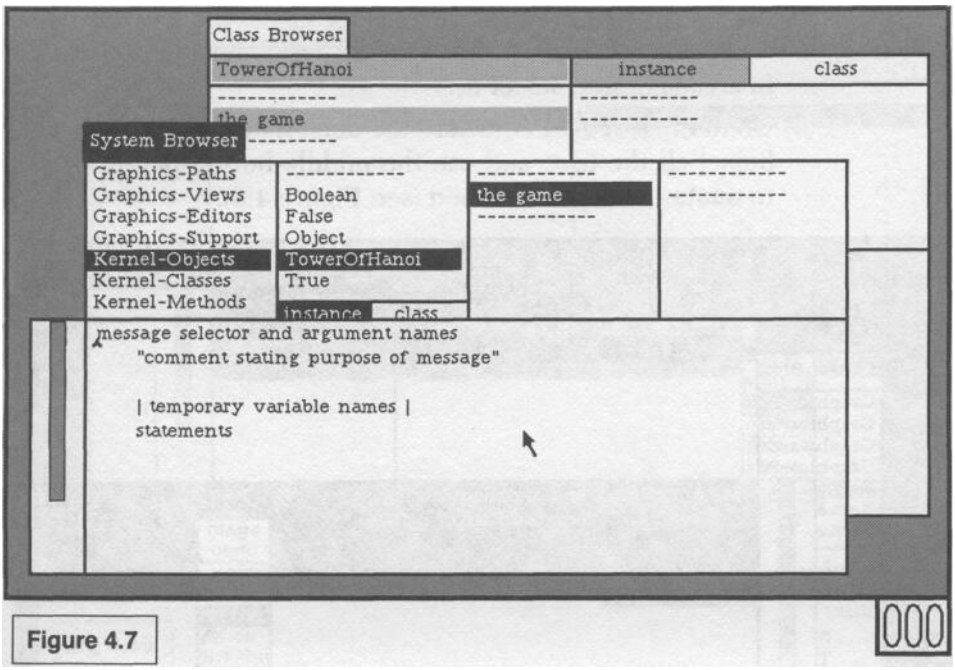

Figure 4.7

entire method at once in area E, you can still select it. Slowly double-click the left button in front of the first character in area E.) Use the middle-button pop-up menu and choose **copy** to copy the method (see Figure 4.8).

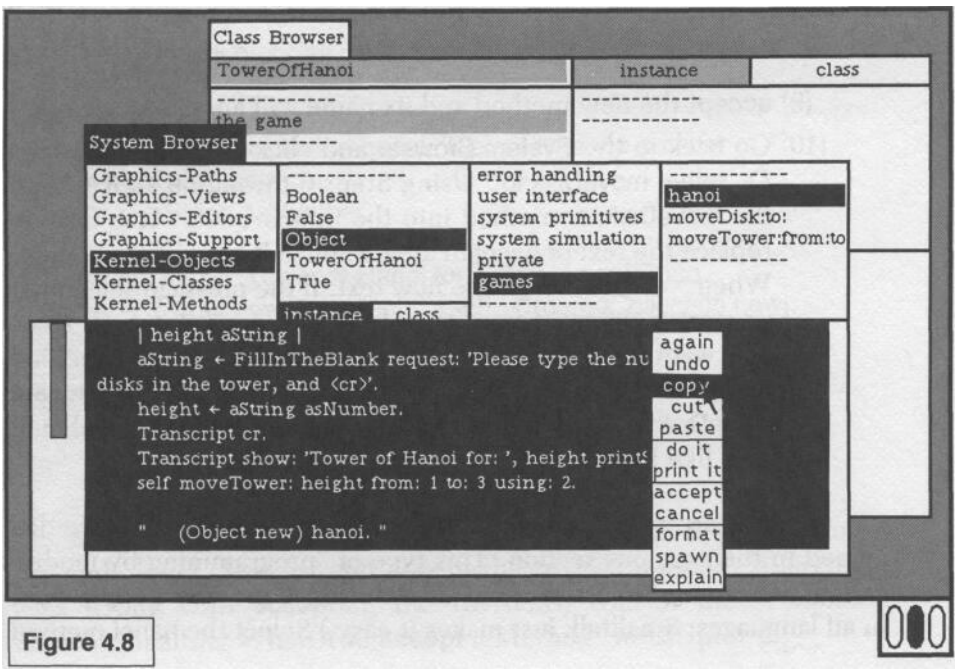

Figure 4.8

(7) Go to the Class Browser (the new browser). Click in it to enter it (wake it up). The category the game should still be selected in area C (if not, select it).

(8) Go to the text area E, where the default method is displayed. Select all the text, and use the middle-button pop-up menu to **paste** the copied method (see Figure 4.9).

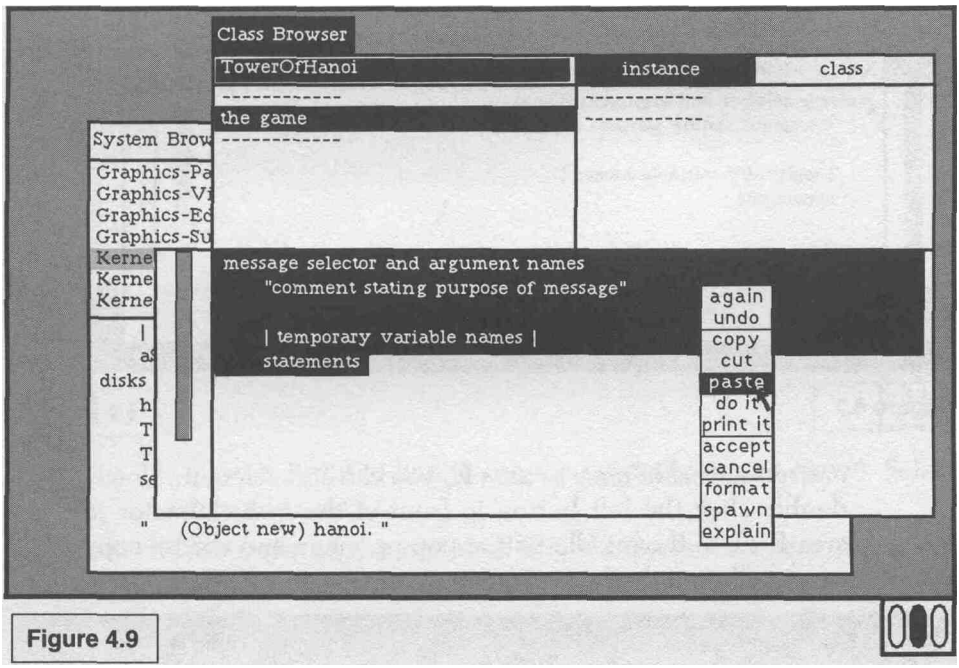

Figure 4.9

(9) **accept** the new method and its name will appear in area D.

(10) Go back to the System Browser and click to enter it. In area D, select moveDisk:to:. Using Steps 6 through 9 above, copy the moveDisk:to: method into the new browser. It is okay to replace the text of hanoi in area E, once it has been accepted. Whenever you accept the new text, if the message selector at its start is different from the old one, the browser will know you want to define a new method (don't copy the method moveTower:from:to:using:). As you are about to **accept** moveDisk:to: in area E of the new browser, the screen should look like Figure 4.10.

Now we are ready to make changes to these methods, as we discussed in the previous section. This type of "programming by modification" is common in Smalltalk. (Indeed; this is the way people do it in all languages; Smalltalk just makes it easy.) Select the hanoi method

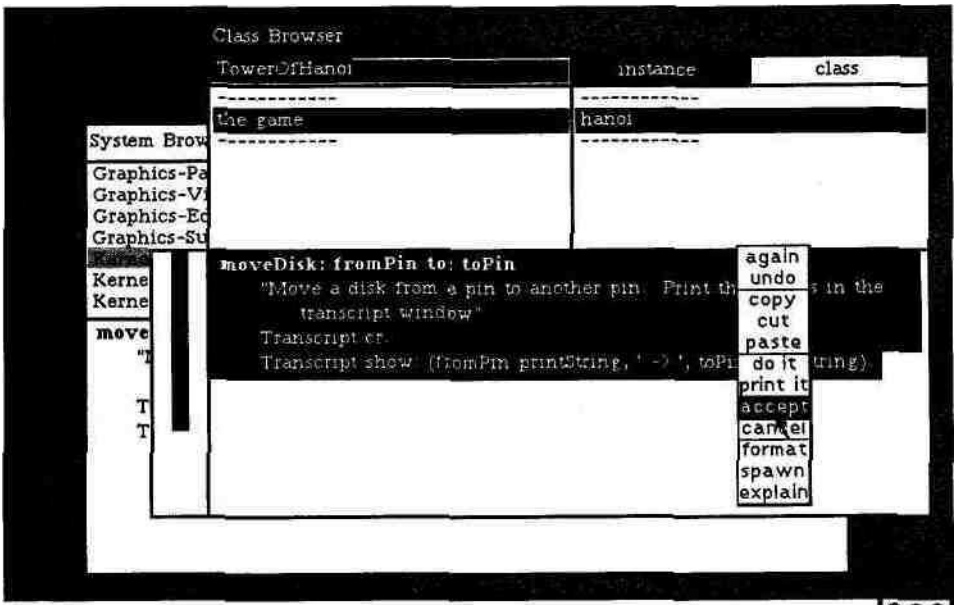

Figure 4.10

in the new browser in area D. Make the underlined changes shown below. (The pocket reference card or Appendix 1 has a refresher on text editing.) Check your changes and **accept** them.

```
hanoi
   "Tower of Hanoi program. Asks user for height of stack of disks"

   | height aString |
   aString <- FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>'.
   height <— aString asNumber.
   Transcript cr.
   Transcript show: Tower of Hanoi for:', height printString.
   stacks <— (Array new: 3) collect: [:each | OrderedCollection new].
   (height to: 1 by: -1) do: [:each | (stacks at: 1) addFirst:
         (Character value: ($A asciiValue) + each -1) ].
   self moveTower: height from: 1 to: 3 using: 2.

         (TowerOfHanoi new) hanoi.
```

After accepting the changes, select moveDisk:to: in area D, make the underlined changes, and accept the revised method. If anything goes wrong with **accepting** the revisions, consult the section on "Troubleshooting When You **accept** a Method" in Chapter 2.

```
moveDisk: fromPin to: toPin
  "Move a disk from a pin to another pin. Print the results in the
    transcript window"
  | disk |
  disk <— (stacks at: fromPin) removeFirst.
  (stacks at: toPin) addFirst: disk.
  Transcript cr.
  Transcript show: (fromPin printString,' -> ', toPin printString,'').
  Transcript nextPut: disk.
  Transcript endEntry.
```

After checking, be sure to **accept.** Notice that you did not copy the method for moveTower:from:to:using: to your new class. This is because TowerOfHanoi is below class Object in the class hierarchy and the version of moveTower:from:to:using: in class Object will do everything we want. TowerOfHanoi inherits all methods from classes above it in the class hierarchy, so Smalltalk will find the existing version. You copied the other messages because we wanted to change them for use in TowerOfHanoi. We will cover inheritance in greater detail in the next chapter.

Run the program by executing:

```
(TowerOfHanoi new) hanoi.
```

(select it in the comment in the hanoi method and then choose **do it.)**

If the program is run with three disks, the System Transcript window should contain:

```
1 ->3A
1 ->2B
3 ->2A
1 ->3C
2 -> 1 A
2 ->3B
1 ->3A
```

If anything untoward happens (as it does so often in this young computer age), turn back to the section on "Troubleshooting Runtime Errors" on page 35 of Chapter 3.

Notice that you can still say (Object new) hanoi and get the same output as you got before we defined TowerOfHanoi. The method that is in control in both programs is moveTower:from:to:using:. Not only is it the same name; it is exactly the same code. How can the same main program get two different results? Different objects are sent the mes-

sage moveTower:from:to:using: and thus inside that method self represents a different object. Object and TowerOfHanoi have separate definitions oimoveDisk:to:, so when moveTower:from:to:using: sends the message, different code runs and the action is different.

The ability to have messages of the same name in two different classes, plus the fact that variables can be of any type, allows programmers to reuse code and to avoid defining the same algorithm in several variations. Because Smalltalk acts differently when the same message is sent to different objects, all code contains an extra level of parameterization, allowing Smalltalk to share code extensively. The existence of only one copy of the code for each algorithm in the system is a major reason for Smalltalk's excellent programming productivity.

RECAPPING THE SMALLTALK TERMINOLOGY

> ***The next best thing to knowing something is knowing where to find it.***
>
> *SAMUEL* JOHNSON

Before you go on to the next example, it might be useful to summarize all the operations you've done so far. Either skip ahead and refer to this section as you have specific questions, or read it now to brush up on the commands.

For each term listed below, the first line gives the name of the operation in Smalltalkese, the next line is the name of that operation in English, the third gives an example showing how to do it, the fourth indicates where examples are found in this book, and the last gives the appropriate sections of the User's Guide. The commands on the menu for text editing (middle-button menu in code windows), such as **cut** and **paste,** are defined in Appendix 1. The format here is:

Smalltalkese
English (well, computerese)
How to
Examples in the text
User's Guide Section

Accept a method
Compile, link, and load a procedure
Choose **accept** in the middle-button pop-up menu in area E of the browser
Pages 24 and 31
11.1 and 11.3

Add a category (also called **add protocol)**
Create a place for a new bunch of procedures
Choose **add protocol** in middle-button pop-menu in area C of the browser
Pages 20 and 50.
11.2

Add a class
Add a new data type
In area B of the browser, deselect any item that is selected by clicking it, then type and accept the new definition in area E
Page 45
12.1 to 12.3

Add a method
Graft a procedure into the Smalltalk system
Find the proper class and category in the browser, then type and accept the new method in area E
Pages 19 to 25
11.3

Browser
The window used to find and modify Smalltalk methods
(Not an action)
Page 16
5.4 and Chapter 9

Choose an item from *a fixed* menu
Choose a command from a stationary list
Move the cursor over the item in the list and click with the left button
Pages 18, 19, and 25
2.3 and 2.4

Choose an item from a *pop-up* menu
Choose a command from a list that "pops up"
Hold down a button (middle or right), move cursor over item, and release the button
Pages 17, 20, and 24
1.3, 2.3, and 2.4

Class
Like a data type, only better
(Not an action)
Page 45
5.1 and Chapter 3 of the Blue Book*

Click
Indicate an item or place
Press and release the left button
Page 17
1.1

Close a window
Remove a window from the display
While in the window, choose **close** from the right-button pop-up menu
Page 35
2.5

Collection
An object that holds a group of objects (Strings and Arrays are collections)
A specific class whose subclasses are the various kinds of collections
Page 46
Chapter 9 of the Blue Book

* As mentioned in the preface, "Blue Book" refers to *Smalltalk-80: The Language and its Implementation* by Adele Goldberg and Dave Robson (Reading, MA: Addison-Wesley, 1983).

Deselect
In a fixed menu, make no item selected
Click on the item that is currently selected
Page 31
2.3

doit
Run the piece of Smalltalk code that is selected, usually a call on a procedure with arguments
Choose **do it** from the middle-button pop-up menu
Page 32
1.3 and 6.1

Editing
Editing (funny, that's the same in English)
The editor is always available in any window you can type into
Pages 20 **and 21**
Appendix 1

Enter a window (also called "Select a view")
Make a window be the active window
Move the cursor into the window and click with the left button
Pages 16 and 31
2.1 and 2.3

File in code
Bring Smalltalk class and method definitions into the system from a file
In the System Workspace, modify a line in the Files section, select it, and choose **do it**
Page 42
22.3

File out code
Save Smalltalk code by writing it on a file in the file system
In areas A, B, C, or D of the browser, choose **file out** from the middle-button menu
Page 41
22.1

Fixed menu
A list of choices that stays on the screen when no button is pressed
(See "Choose an item from a fixed menu")
Pages 18 and 24
2.3

Frame a window
Change the shape and/or location of a window
Choose **frame** from the right-button pop-up menu (see example in the text)
Page 51
1.3 and 2.2

Instance
One of the objects described by a class; it has memory and responds to messages
(Not an action)
Page 45
5.1 and Chapter 1 of the Blue Book

Message
A request for an object to carry out one of its procedures
(See "Send a message")
Page 12
5.1 and Chapter 1 of the Blue Book

Message selector
The name of a procedure (or of an operator)
(Not an action)
Page 11
Chapter 2 of the Blue Book

Method
A procedure—that is, the code that runs when a message is sent; it can also be thought of as a description of one of an object's operations
(Not an action)
Page 9
5.1 and Chapter 1 of the Blue Book

Object
A package of data and procedures that belong together; the contents of any variable or constant is an object
(Not an action)
Page 11
5.1 and Chapter 1 of the Blue Book

Pop-up menu
A list of choices that appears when you press the middle or right button
(See "Choose an item from a pop-up menu")
Pages 20 and 24
1.3 and 2.3

Receiver
The object that was sent the current message
(See self below)
Page 12
5.4 and Chapter 1 of the Blue Book

Scroll a window
Move the contents of a window to see another part of it
(See example and pocket reference card)
Page 17
2.3

Send a message
Call a procedure—that is, ask an object (the receiver) to do an operation with some arguments and return a result
(Happens inside methods)
Page 12
5.1 and Chapter 1 of the Blue Book

Select an item in *a.fixed* menu
(Means the same thing as "Choose an item from a Bxed menu")
Move the cursor over the item in the list and click with the left button
Pages 18, 19, and 24
2.3 and 2.4

Select an item in a *pop-up* menu
Move between the choices in a pop-up menu, but do not release the button
Hold down the middle or right mouse button and move around in the menu until the desired item shows in reverse video
(Not mentioned in the text, but part of choosing an item from a pop-up menu)
2.3

Select text
Indicate a piece of text upon which the next editing operation will act
Hold down the left mouse button, move until the text is reverse video, then release
Page 23
3.1

self
The local name for the object that received the message whose method we are in
Used to send another message to the same object that received this one
Page 13
5.4 and Chapter 1 of the Blue Book

Spawn a new browser
Ask for a specialized browser at another place on the screen
Choose **spawn** from a middle-button pop-up menu in any part of the browser (in License 1, choose **browse)**
Page 51
9.4