# 7

# ON YOUR OWN

## FINDING WHAT YOU NEED

***The best effect of any book is that***
***it excites the reader to self activity.***

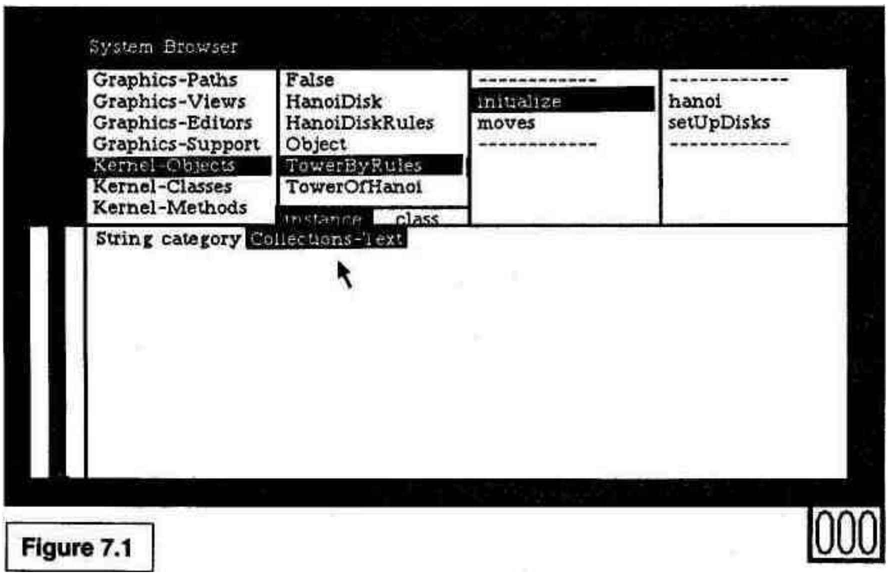
THOMAS CABLYLE

Before you further extend your program, write other programs, and become an expert at the Smalltalk system, you should know a few hints. As we said before, each Smalltalk class is like a "module" or "package," and the entire Smalltalk-80 system is like a large subroutine library. The browser is an information retrieval window into that library. Classes are grouped into "categories" to make them easier to find. Some categories contain classes which are related by all being subclasses of a particular class (the subclasses of Number in **Numeric-Numbers,** for example). In general, a common purpose or a common use unites the classes in a category, and while some of them are subclasses of each other, many are not.

You are not expected to memorize where to find classes, or which classes respond to which messages. However, the **Kernel", Numeric-,** and **Collections-** categories are so important that we urge you to read Chapters 6 through 14 in the Blue Book to learn what these classes do and where to find them in the browser.

When you sit down to "write a program" in the conventional sense, you will almost always define a new class. When you add functionality to the system, or modify the behavior of the system, you will usually add methods to existing classes. Another common practice is to make a subclass of an existing class, solely to override a couple of its methods, being confident that you won't interfere with the normal functioning of that class.

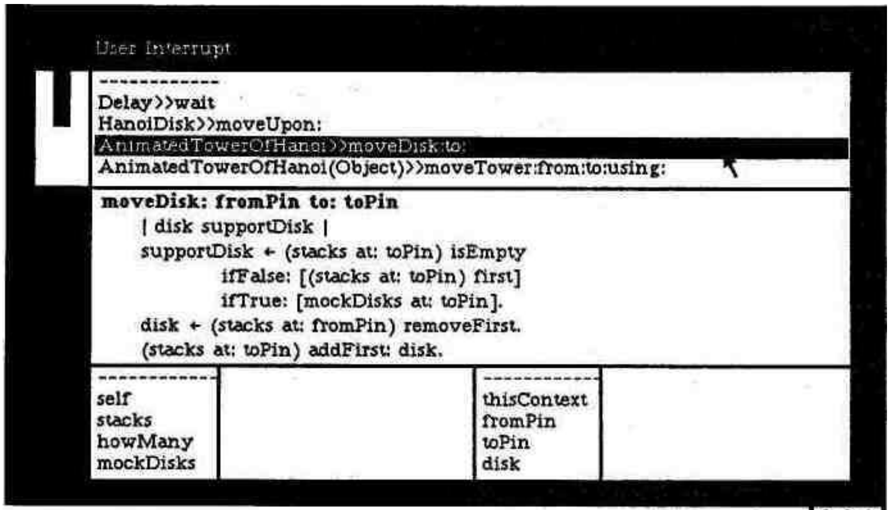
As a Smalltalk programmer you rarely just sit down and write code. A common practice is to find another part of the system that does something very similar to the thing you want to do. You read that code, copy it into the method you are writing, and modify it to your specific needs. There are several techniques for finding a piece of code when you see its actions in another part of the system. Let's try some of these techniques. Suppose you know the name of class String, but don't know which category it is in. Enter area E of a browser or the transcript. Type String category and choose **print it** from the middle-button menu. The answer, Collections-Text, is inserted in the text just after your selection, as shown in Figure 7.1. (You will need to **cancel** the change to the text before the system will let you try the other examples here.)



```
String category Collections-Text
```

Figure 7.1

Another technique to find a piece of code is to interrupt the system when it is doing the thing you want to find out about. Suppose you can run the Tower of Hanoi animation, but don't know where to find the code in the browser. You can start the hanoi animation and then type *control C (Command period* on a Macintosh). A new window



```
------------
Delay>>wait
HanoiDisk>>moveUpon:
AnimatedTowerOfHanoi>>moveDisk:to:
AnimatedTowerOfHanoi(Object)>>moveTower:from:to:using:
```

```
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.
```

```
------------        ------------
self                thisContext
stacks              fromPin
howMany             toPin
mockDisks           disk
```

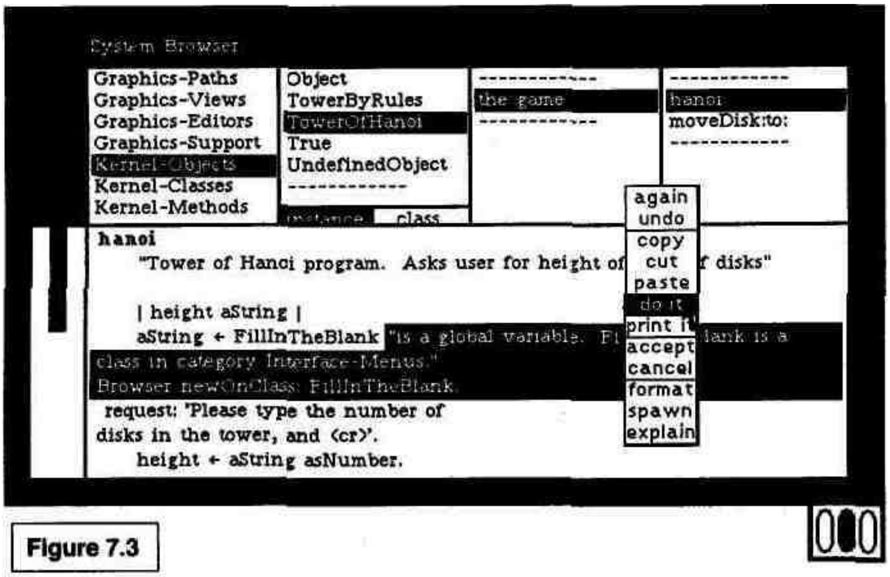
Figure 7.2

will appear. If you open the error window by choosing **debug** from the middle-button menu, you will see the execution stack in the upper pane. You can find the code you are after by clicking on the various message names in the upper pane (see Figure 7.2). The debugger is extremely useful, but we haven't the space to cover it here. See Chapter 19 of the User's Guide for complete instructions.

When you are reading a specific piece of code, you will want to know more about the variables and messages you find. Suppose you are looking at the code for the method hanoi in the browser. You see a reference to something called FillInTheBlank, but you have no idea what it is. Select FillInTheBlank and then choose **explain** from the middle-button menu. The system will tell you about the thing you have selected. **explain** works for any single token in a method, including variables, selectors, pieces of selectors, and all punctuation, (**explain** has been taken out of Apple's Level 0 image to save space.) Sometimes the description includes a Smalltalk expression, which is meant to be evaluated, as shown in Figure 7.3.

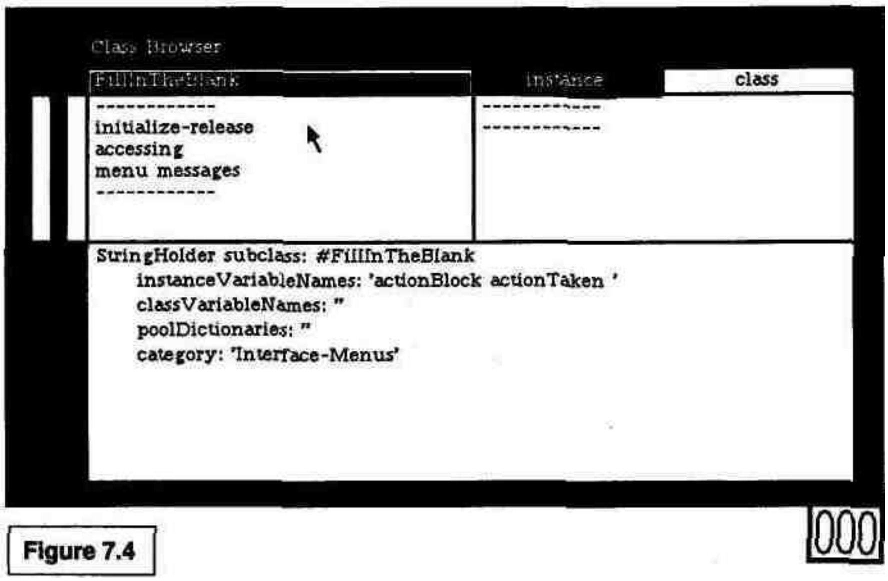
When you say **do it,** the system asks you to frame a new browser that is specialized to the variable or class. You can now explore class FillInTheBlank and see what messages it responds to (see Figure 7.4).

Suppose that you are exploring a class and are trying to discover what an instance variable in that class does. For example, take the instance variable stacks in the class AnimatedTowersOfHanoi. In area B of the browser, select AnimatedTowersOf Hanoi. The first thing to do is to choose **comment** and read it. If the comment is not very illuminat-



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

    | height aString |
    aString ← FillInTheBlank "is a global variable.  FillInTheBlank is a
class in category Interface-Menus."
Browser newOnClass: FillInTheBlank.
 request: 'Please type the number of
disks in the tower, and <cr>'.
    height ← aString asNumber.
```

Figure 7.3

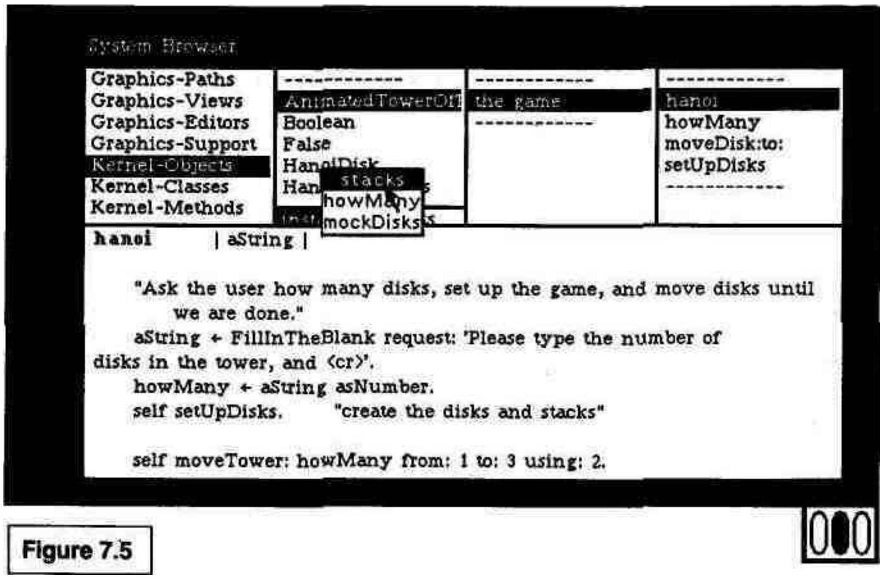


```
StringHolder subclass: #FillInTheBlank
    instanceVariableNames: 'actionBlock actionTaken '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Interface-Menus'
```

Figure 7.4

ing, you may want to see every place in the system that the variable stacks is used. In area B, choose **inst var refs** from the middle-button menu. The browser puts up a new menu that lists the instance variables by name. Click on **stacks,** as shown in Figure 7.5.

The system asks you to frame a window for a new browser on a list of methods. This browser shows you all the methods in the system in which the instance variable stacks appears. You can see the method in the lower pane by clicking on its name in the upper pane. You can

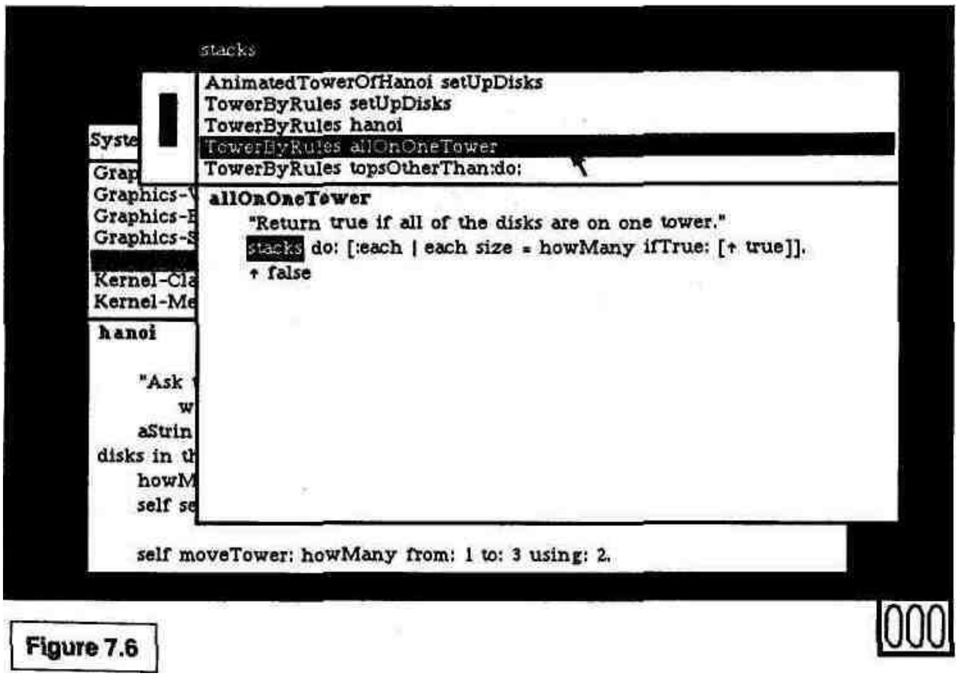


```
hanoi        | aString |

    "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.
```

Figure 7.5

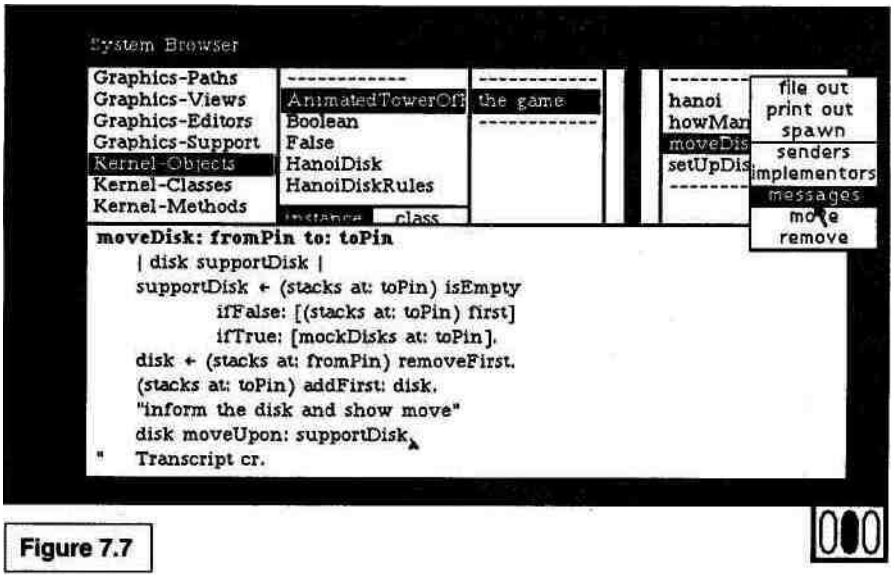
make changes to the method and **accept** it without leaving this method list window. Whenever you need to change all uses of a variable systematically, this kind of window is ideal (see Figure 7.6).

The item **class var refs** in the middle-button menu is exactly like **inst var refs.** It shows you all the places a particular class variable is used.


stacks
AnimatedTowerOfHanoi setUpDisks
TowerByRules setUpDisks
TowerByRules hanoi
TowerByRules allOnOneTower
TowerByRules topsOtherThan:do:

```
allOnOneTower
    "Return true if all of the disks are on one tower."
    stacks do: [:each | each size = howMany ifTrue: [↑ true]].
    ↑ false
```

Figure 7.6

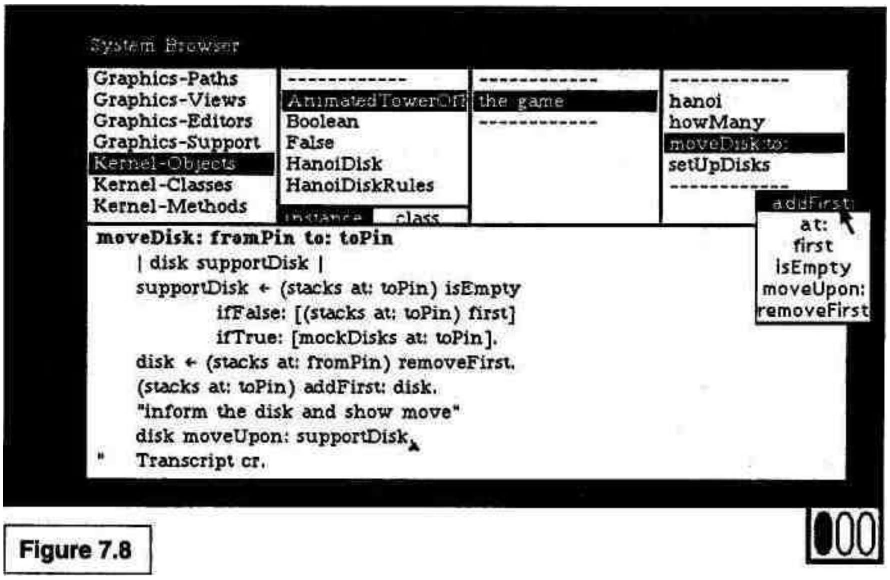


```
moveDisk: fromPin to: toPin
    | disk supportDisk |
    supportDisk ← (stacks at: toPin) isEmpty
            ifFalse: [(stacks at: toPin) first]
            ifTrue: [mockDisks at: toPin].
    disk ← (stacks at: fromPin) removeFirst.
    (stacks at: toPin) addFirst: disk.
    "inform the disk and show move"
    disk moveUpon: supportDisk
"   Transcript cr.
```

Figure 7.7

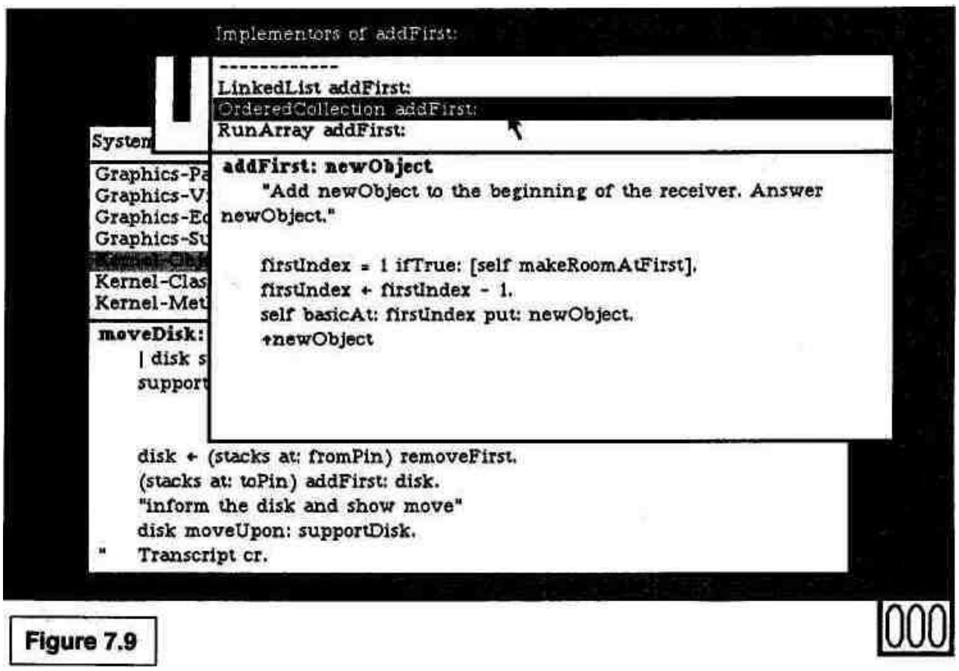
As you read code, you will often come across an unfamiliar message name. Suppose you are looking at the method moveDisk:to: in area E of the browser, and you notice the selector addFirst:. To see the code that implements addFirst:, go to area D and choose **messages** from the middle-button menu (see Figure 7.7).

A new menu will appear with the names of all messages sent from moveDisk:to: (see Figure 7.8). Click on addFirst:.



```
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.
```

Figure 7.8



```
------------
LinkedList addFirst:
OrderedCollection addFirst:
RunArray addFirst:
```

```
addFirst: newObject
    "Add newObject to the beginning of the receiver. Answer
newObject."

    firstIndex = 1 ifTrue: [self makeRoomAtFirst].
    firstIndex ← firstIndex - 1.
    self basicAt: firstIndex put: newObject.
    ↑newObject
```

```
moveDisk:
    | disk s
    support

    disk ← (stacks at: fromPin) removeFirst.
    (stacks at: toPin) addFirst: disk.
    "inform the disk and show move"
    disk moveUpon: supportDisk.
"   Transcript cr.
```

Figure 7.9

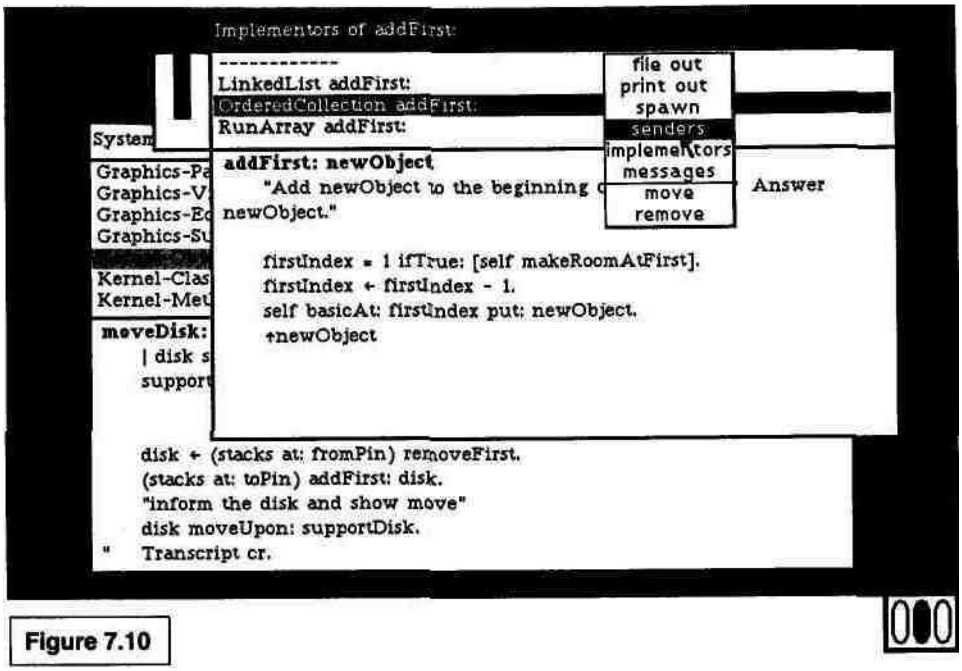
The system will ask you to frame a special browser containing all the methods in the entire system that are named addFirst: (see Figure 7.9).

Notice that the classes LinkedList and RunArray have specialized responses to the message addFirst:. Suppose you want to see another call on addFirst: in order to really understand how it works. Click on OrderedCollection addFirst:, as shown in Figure 7.9. Now hold down the middle-button in the upper part of the window, and choose **senders** (see Figure 7.10).

**senders** finds all methods in the system which call addFirst:, and the system asks you to frame a window in which to show them (see Figure 7.11).

The list of senders of addFirst: includes calls on all three of the implementations (definitions in different classes), since the system cannot determine the class of the receiver of the message until runtime. Any message list menu, like area D of the browser, allows you to choose **messages** and **senders** from its middle-button menu. See the Inquiry section in the System Workspace window for some useful ready-made templates for searching the system. (Also see Section 5.4 and Chapter 10 of the User's Guide.)

In the Tower of Hanoi program, the method moveTower:from:to:using: is not in the class AnimatedTowersOfHanoi. It is inherited from Object because TowerOf Hanoi is a subclass of Object and



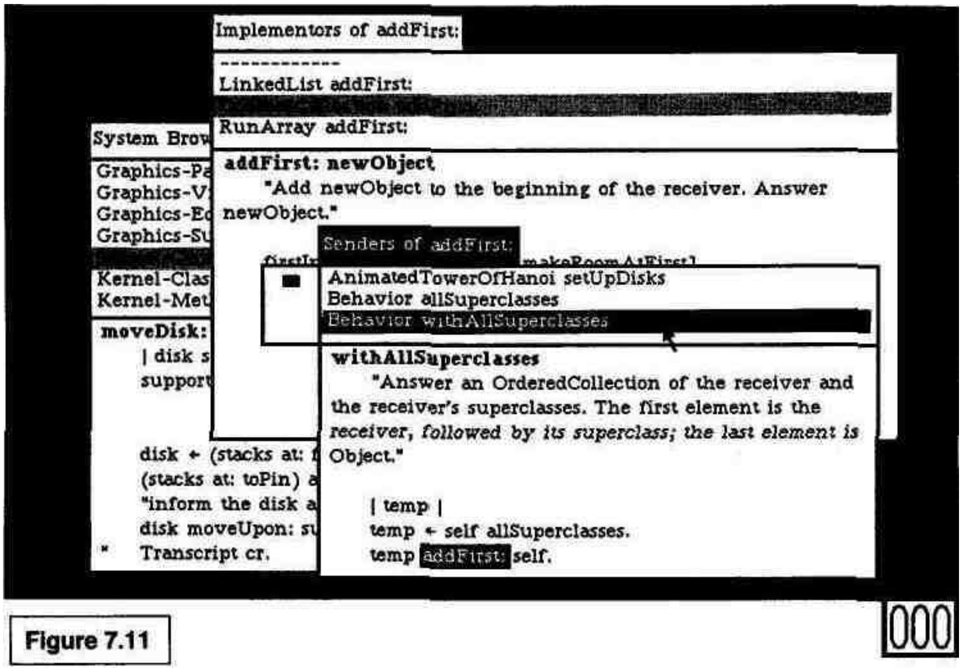
```
------------
LinkedList addFirst:
OrderedCollection addFirst:
RunArray addFirst:
```

```
addFirst: newObject
    "Add newObject to the beginning c                Answer
newObject."

    firstIndex = 1 ifTrue: [self makeRoomAtFirst].
    firstIndex ← firstIndex - 1.
    self basicAt: firstIndex put: newObject.
    ↑newObject
```

```
moveDisk:
    | disk s
    support

    disk ← (stacks at: fromPin) removeFirst.
    (stacks at: toPin) addFirst: disk.
    "inform the disk and show move"
    disk moveUpon: supportDisk.
"   Transcript cr.
```

Figure 7.10

AnimatedTowersOfHanoi is a subclass of TowerOfHanoi. When you are exploring a class, it is valuable to see the subclass relationships. Enter the browser, and make sure AnimatedTowersOfHanoi is selected in area



```
------------
LinkedList addFirst:
RunArray addFirst:
```

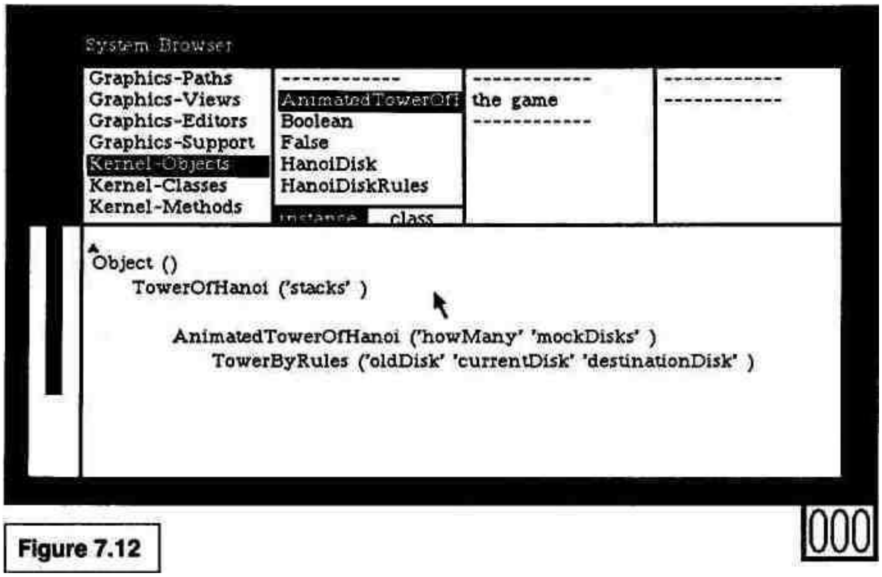
```
addFirst: newObject
    "Add newObject to the beginning of the receiver. Answer
newObject."
```

```
AnimatedTowerOfHanoi setUpDisks
Behavior allSuperclasses
Behavior withAllSuperclasses
```

```
withAllSuperclasses
    "Answer an OrderedCollection of the receiver and
the receiver's superclasses. The first element is the
receiver, followed by its superclass; the last element is
Object."

    | temp |
    temp ← self allSuperclasses.
    temp addFirst: self.
```

```
moveDisk:
    | disk s
    support

    disk ← (stacks at: f
    (stacks at: toPin) a
    "inform the disk a
    disk moveUpon: su
"   Transcript cr.
```

Figure 7.11



```
Object ()
    TowerOfHanoi ('stacks' )

        AnimatedTowerOfHanoi ('howMany' 'mockDisks' )
            TowerByRules ('oldDisk' 'currentDisk' 'destinationDisk' )
```

Figure 7.12

B. Choose **hierarchy** from the middle-button menu. Area E will change into a picture of the subclass hierarchy (see Figure 7.12).

A system with overlapping windows and the ability to copy code from one window to another allows you to adopt a style of work different from a "full screen" editor on a character display. While writing code you may want three browsers (or spawned windows) open at once. One contains the method you are currently writing, one is for looking at the rest of your own code (the methods you just wrote), and one is for browsing around the system to find things to copy or use.

To open a new browser, move the cursor so that it is in the gray area between windows, and choose **browser** from the middle-button pop-up menu. (As you might guess, **file list, workspace, system transcript,** and **system workspace** on the same menu create new instances of other windows. If you ever accidentally hit the right button of the mouse and close a window unintentionally, you can use these menu items to get back another window of the same kind.)

Since this book is a slim introductory volume, we have not covered a number of important topics in Smalltalk. Here is a brief list of topics, so that you will experience "name recognition" when you hear about them later, or see them in the Smalltalk system.

- You can send a message to a class, instead of to one of its instances. Messages to a class are most often used to create instances or to ask about class-wide information. Some examples we have seen are Array new:, FillInTheBlank request:, and Character value:. You can view class messages by clicking on the word **class** at the bottom of area B in the browser. See Chapter 5 of the Blue Book.

- Besides local variables and instance variables, there are three other kinds of variables. Global variables are stored in a dictionary called Smalltalk. You are able to mention the name of a class in a method because all classes are stored under their names as globals. The globals that are not classes (and thus cannot be found in the browser) are listed in the Globals section of the System Workspace window. We have seen "class variables," such as DiskGap in HanoiDisk. They are common to a particular class, its subclasses, and all of their instances. Pool variables are a particularly obscure kind, shared by classes that are not related by subclassing. See Chapter 3 of the Blue Book.
- We have not explained subclasses, superclasses, and inheritance in their full glory. See Chapter 4 of the Blue Book. In some versions of Smalltalk, a class can even inherit behavior from more than one direct superclass.
- Inspector windows are useful because they let you look inside objects. You can send the message inspect to any object, or you can choose **inspect** from the middle-button menu in any window that is a list of objects. Lists of objects appear in the bottom panes of the debugger and also in inspect windows. Once in an inspector, you can change the values of the instance variables of the object.
- The System Workspace is simply a window full of text. You may have used it already to bring in source code from a file. The text in the window is a series of templates for commonly used actions. You edit an expression to customize it, select it, and then choose **do it** or **print it.** (do **it** executes the code and **print it** both executes the code and inserts the result in line.) The commands are for interacting with an external operating system (the Files section), saving your work (fileOutChanges), searching the system (the Inquiry section), managing Smalltalk's global resources (Display and Processor), and making measurements (coreLeft, spyOn:). See Section 6.3 and the System Workspace Index of the User's Guide.
- "Projects" are multiple screens that you can flip among. They are useful for separating different projects you are working on. You can set up a screen full of windows for each project you are working on (such as writing a paper, reading your mail, and designing a new browser), and then flip quickly between them. See Chapter 4 of the User's Guide.

- There are four different tools for managing programs you have written. You can save the entire state of your system by choosing **save** or **quit** from the middle-button menu in the gray space between windows. You can write out the source code for individual classes (**file out** in the middle-button menu in area B of the browser), or all your changes (in the System Workspace). Or, you can use the Change-Management Browser for a high degree of control over your changes.
- Yes, we even offer insurance! If your machine crashes, you can always get the text of all the methods you wrote before the crash by restarting the system and then executing (Smalltalk recover: 10000). For the specifics, read Chapter 23 of the User's Guide.

## WHAT IS OBJECT-ORIENTED PROGRAMMING, REALLY?

The series of example programs we have built to solve the Tower of Hanoi puzzle represents a spectrum of style. The first version (in class Object) was simply a transcription of the normal recursive solution. Even though the Smalltalk version used objects and sent messages to them, it was not an example of object-oriented programming. It is perfectly possible to write the exact analog of FORTRAN programs in Smalltalk. When we divided the problem up into disks and a game-wide object, the algorithm became more object-oriented. The algorithm used in AnimatedTowerOfHanoi and HanoiDisk was not totally object-oriented, however, because the recursive version could compute the solution without using the stacks or the data in the disks! The rule-based algorithm really uses its data structures. In class TowerByRules and HanoiDiskRules, the objects use their data structures and decide where to move by sending messages to each other. In the last chapter we wrote a program that is truly object-oriented.

The idea of modular code is well established in computer science, but it alone does not make a program object-oriented. In a language other than Smalltalk, the routines that deal with a specific function are packaged together. The data structures, however, are usually global. In a typical text editor, it is common for all routines that actually modify the text to be in one package, but usually the data structures that hold the actual text are global. Any other routine *could* reach in and change text. In real-world systems, "reaching in the back door" to modify the data structure is quite common. Suppose segments of text in the text editor have "dirty bits" associated with them. This bit needs to be turned on every time a piece of text is modified or the changes will not be saved to disk. The only way a programmer can be sure that the dirty bit is being set every time it should be is to search the entire program for places that write into the textual data structures. In Smalltalk, if an object owns some data, no other object can reach that data without sending a message to the owner. If an instance of class Text needs to enforce the setting of dirty bits, the programmer must search only the methods in class Text.

Let's get away from talking about protection of data. Imagine that every fragment of program and every piece of data are floating together in space. Now imagine pieces of string between code and the data it uses, and between segments of code that are used together or perform a similar function. Let's move everything around until the total amount of string is the smallest. Clustered around each data structure are the routines that use it. Natural divisions in the code become apparent. When a problem is segmented into classes along natural boundaries, the resulting program is beautiful. Dividing a problem up into objects is a process of putting things where they belong.

The most important part of object-oriented programming is not any technical advantage it gives, but the fact that it crosses a threshold of perception. When we put all of the information associated with a disk into class HanoiDisk, we didn't just clean up the code in AnimatedTowerOfHanoi. We allowed ourselves to think of that body of information and action as a single unit, namely, "a disk." We are used to perceiving the world around us as made up of "objects," and our brains arrange information into "chunks." By using objects in a programming language, we can tap into an existing convention. Thinking of an algorithm in terms of objects makes it easier to understand. This ease of understanding often comes not from the details of the class you are working on, but from *not* having to think about the rest of the program. When you are working inside one class, you are largely safe from the side effects and complexities of parts of your program outside that class.

Dividing a problem up into objects and defining actions that are "natural" for those objects actually make programs simpler. When the actions of an object are divided into the right kinds of "chunks," we can think and write code at a higher level. We know that the methods we call from the high-level code are correct, because they match the way we think about the problem. When that trust exists, we make fewer mistakes. We carry around less tension about whether the subordinate procedures are doing the right thing. Programs are uncluttered and easier to maintain.

In Smalltalk, the overhead of sending a message is small. Since sending a message is the only thing you can do in a piece of Smalltalk code, it has been optimized. An average method has ten to fifty message sends in it. If an expression keeps cropping up over and over in a program, it probably deserves to be a separate method. The goal in Smalltalk is to make creating a new method as easy as possible. Writing a new method and calling it in the original program should be easier than defining a new procedure in Pascal. Declarations and preambles are very short in Smalltalk, and you can edit, run, and debug without changing environments. Processing a file through a compiler after typing a program into it introduces unnecessary complexity. Why should a file system come between you and your program? Messages and methods are meant to be lightweight and quick to install. In Smalltalk, compiling a method and linking it into the system take only a few seconds, as opposed to several minutes in a "batch-processed" Pascal or C system where many modules may have to be recompiled.

Some of the signs of non-object-oriented programming are too many loops and too much indexing of multi-level data structures. Compare the method for bestMove in HanoiDiskRules (shown below) with the equivalent code in a style that is not object oriented.

```
bestMove      | secondBest |
   "If self can move two places, which is best? Return the top disk of
      the pole that this disk has not been on recently."
   TheTowers polesOtherThan: self do: [:targetDisk |
      width < targetDisk width ifTrue:
         [secondBest <- targetDisk.
         targetDisk pole = previousPole ifFalse: [ f  targetDisk]]].
    t  secondBest "as a last resort, return a pole it was on recently"
```

If we rewrite this program in the style of a traditional language, the code becomes bulky and hard to understand. polesOtherThan:do: is not written as a separate procedure, but this is probably how most Pascal programmers would write it.

```
bestMove: movingDisk on: currentPole
   "If movingDisk can move to two places, which is best? Return
      the pole that the disk fits on and has not been on recently."
   1 to: 3 do: [:targetPole | "each other pole"
      targetPole ~= currentPole ifTrue:
         ["Not the current pole"
            (stacks at: targetPole) isEmpty
            ifTrue: ["We know it fits on an empty pole"
               secondBest <- targetPole.
      "Is this the pole we just came from? Look in an array of
        previousPoles"
      targetPole = (previousPoles at: movingDisk) ifFalse:
        [ t targetPole "the best"]]
    ifFalse: ["a real disk is on top"
      "compare the widths"
      movingDisk < ((stacks at: targetPole) at:
                (stackTops at: targetPole))
        ifTrue:["itfits"
          "Is this the pole we just came from?
            Look in an array of previousPoles"
          targetPole = (previousPoles at: movingDisk) ifFalse:
            [ f targetPole "the best"]]]]].
f secondSest        "as a last resort, return the pole it was on recently"
```

Good design and clean code are not the sole province of Smalltalk. It is possible to write extremely beautiful code in other languages, as well as terrible code in Smalltalk. Smalltalk even has its own particular kinds of brier patches. (Look at the code in Behavior, ClassDescription, and Class that runs when you accept a class definition for an existing class in which you rename its instance variables.) The designers of Smalltalk believe that you already know what makes a good design and clean code. Smalltalk tries to encourage you to follow your instincts for good design.

Now that you've successfully created a truly object-oriented animated program, consider yourself an Official Level One Smalltalk Programmer! Perhaps the next step is to add some bells and whistles to your animated Tower of Hanoi program. Appendix 4 has seven suggestions (four bells and three whistles), with Appendix 5 hinting at the answers and Appendix 6 just giving them away. If you want to learn even more about Smalltalk, we suggest going through the User's Guide (the bit-editor is really fun). You might then read Part I of the Blue Book, and try the FinancialHistory example in Appendix I of the User's Guide. You've had a taste of Smalltalk, and we hope to have whetted your appetite. Bon appetit!