APPENDIX 1

THE SMALLTALK TEXT EDITOR

To replace a passage of text, select it by pressing the left button at the beginning of the passage and releasing it at the end. Then type the new passage. The first keystroke will delete the old passage.

The middle-button pop-up menu contains the commands used to edit text. This menu is available wherever you can type text.

again do the last **paste** again, but in a new place. Find the next occurrence of the text that was pasted over last time. Replace that text.

undo undo the last editing action (only works one command back and only if the selection has not moved).

copy remember the text that is currently selected.

cut remove the text that is currently selected.

paste replace the selection with what was last cut, copied, or typed. *

do it treat the current selection as Smalltalk code and evaluate it.

print it treat the current selection as Smalltalk code, run it, and insert the result after the selection.

* Macintosh users should note that paste will not paste in the last thing typed. It must have been cut or copied. In this respect, the text editor in Apple's version of Smalltalk has been modified to be like the Macintosh text editor.

accept compile, link, and load the method (or class definition) in this window.

cancel redisplays the text as it was at the time of the last **accept** (undoes all edits since the last **accept).**

format pretty print the text for this method; in other words indent the program so it is easy to read. If you like the new form, choose **accept** afterwards. Does not work if you have changed the text since the last **accept.**

Spawn creates a new browser, just for this method.

explain inserts an explanation of the single thing that is selected. It has trouble if more than one "thing" is selected.

For more detail on the text editor, see Chapter 3 of the User's Guide.

APPENDIX 2

HOW TO TALK TO YOURSELF WHEN READING' SMALLTALK

As we mentioned above, some people feel the need to pronounce when writing programs. We have provided a Smalltalkese reading of moveTower:from:to:using: and moveDisk:to:.

```
moveTower: height from: fromPin to: toPin using: usingPin
  "Recursive procedure to move the disk at a height from one
    pin to another pin using a third pin"
  (height > 0) ifTrue: [
    self moveTower: (height-1) from: fromPin to: usingPin using: toPin.
    self moveDisk: fromPin to: toPin.
    self moveTower: (height-1) from: usingPin to: toPin using: fromPin]

  "This comment gives an example of how to run this program. Select
  the following and choose 'do it' from the middle-button menu.
    (Object new) moveTower: 3 from: 1 to: 3 using: 2
```

The method for move-tower-from-to-using. The arguments are height, from-pin, to-pin, and using-pin. (A recursive procedure to move the disk at a height from one pin to another pin using a third pin.) Height is greater than zero, if true, send yourself move-tower with height minus one, from from-pin, to using-pin, using to-pin. Send yourself move-disk from from-pin to to-pin. Send yourself move-tower with height minus one, from using-pin, to to-pin, using from-pin. Return self ("return self" is the "amen" of Smalltalk). This benediction is implicitly at the end of every method.

```
moveDisk: fromPin to: toPin
  "Move a disk from a pin to another pin. Print the results in the
    transcript window"
  Transcript cr.
  Transcript show: (fromPin printString,' -> ', toPin printString).
```

The method for move-disk-to. The arguments are from-pin and to-pin. (Move a disk from a pin to another pin. Print the results in the transcript window.) Transcript carriage return. Transcript show from-pin's print string, concatenated with the string for a little arrow, concatenated with to-pin's print string. (This program is not actually doing anything about moving the disks!) Return self (Amen).

APPENDIX 3

METHODS MISSING FROM THE APPLE LEVEL 0 IMAGE

Early versions of the Level 0 Smalltalk system for the Macintosh 512K have some methods missing. The Level 0 system is a cut-down version of Apple's Level 1 system (which is for machines with a megabyte of memory or more). A few classes and many messages were removed to make a small system. The programs in this book happen to use two methods that were taken out, as well as one that was changed. Please follow the directions below to install the missing methods, and then return to denning the method hanoi in Chapter 3.

(1) Enter area A of the browser and scroll to the category **Interface-Browser.** It is above **Kernel-Objects** and is the fourth **Interface-** category. Select **Interface-Browser** by clicking on it.

(2) Select MessageCategoryListView in area B. (Area B may not be wide enough to see all of the name. Of the two names that begin MessageCategoryLi..., select the second one.)

(3) In area C, the system automatically selects **As yet unclassified.** Choose list: in area D.

(4) In area E, all you need to do is add the word self and a space to the beginning of the last line. The change is underlined below.

```
list: anArray
   "Refer to the comment in ListView|list:"

   super list: anArray.
   (anArray ~= nil and: [anArray size =1]) ifTrue:
      [Selection <-1.
      self controller preSelectModeSelection: 1]
```

(5) Choose **accept** from the middle-button menu. (You may be wondering what you just fixed. Notice that in Step 3 above, the single item in area C was selected automatically. The bug we just fixed was introduced when that feature was added. When we create a new browser window, as is done in Chapter 4, this code tries to select the only item in area C before the variable controller is initialized. Sending the message self controller instead gets us the same variable, but the code happens to check if it is uninitialized. But wait, we don't yet know enough about Smalltalk to make sense of this.)

(6) Enter area A of the browser and select the category **Interface-Menus. It** is the category above **Interface-Browser,** the one we were just in. Select **Interface-Menus** by clicking on it.

(7) Select FilllnTheBlank in area B.

(8) Earlier we said that we would never use the **class** switch in area F of the browser (below area B). Well, now we have to use it just for a moment, and then we will switch it back to **Instance.** Move the cursor down from area B to area F and click on class.

(9) In area C, the system automatically selects **As yet unclassified.** Look in area D to see if the method request: is there. If it is, you don't have to type it in after all, and can go directly to step 13. Otherwise . . .

(10) In area E, select all the text and replace it with

```
request: messageString
    "Create an instance of FilllnTheBlank whose question is mes-
     sageString. Display it centered around the cursor. Return the
     string that the user types and accepts."

    self
        request: messageString
        displayAt: Sensor cursorPoint
        centered: true
        action: [response | response]
        initialAnswer: ".   "< - two single quotes"
     f  response
```

(11) In the line before the last line, initialAnswer: ". has two single-quote characters after the colon. Two single quotes in a row is a null String. It is the same thing as (String new: 0). (We also can't resist telling you what this code does. self is the object FilllnTheBlank, which is a "class." We will learn about classes in Chapter 4. self is sent the message

request:displayAt:centered:action:initialAnswer:. Because of the block, the local variable response is set as a side effect. In the last line, the method returns the value in the variable response to the caller. We will discuss return in detail later.)

(12) Choose **accept** from the middle-button menu.

(13) Move to area F and click on **instance.** *Be sure to do this!* If you leave the switch on **class,** you won't be able to Bnd things in the browser. Now let's define the other missing method.

(14) Enter area A and scroll to the category **Collections-Text.** It is above the **Interface-** categories and is the fourth **Collections-** category. Select **Collections-Text.**

(15) Select String in area B.

(16) In area C, the system automatically selects **As yet unclassified.** If asNumber is already in area D, you can skip to Step 19.

(17) In area E, select all the text and replace it with

```
asNumber
  "self is a string with the ASCII characters for some digits.
  Convert the digits to a number and return it."

  f Number readFrom: (ReadStream on: self)
```

(18) Choose **accept** from the middle-button menu. (Both the Apple Level 0 and Level 1 systems are Xerox License 1 systems. If you have a License 2 system and are reading this section anyway, we must tell you that License 2 has a new name for the message on:. In the code for asNumber, you will find ReadStream onCoilection: self instead of ReadStream on: self.)

(19) Scroll back to **Kernel-Objects.** It is below all the **Interface-** categories. Select **Kernel-Objects** in area A, Object in area B, **games** in area C, and continue with the example on page 38 of the text.

APPENDIX 4

EXERCISES

No *more training do you require.*
Already know you that which you need.
YODA in *The Empire Strikes Back*

To get more experience, modify the animated Tower of Hanoi program to add some bells and whistles. Here are a few suggestions. Appendix 5 contains hints to help you, and Appendix 6 gives some example solutions.

(1) The disks in the animated example are black. Change them to gray.

(2) The disks move from one stack to another by moving directly from their old positions to their new places. Change this so that a disk jumps up above its original stack, jumps across to the new stack, and then jumps down to its final position.

(3) Make the animation pause when any mouse button is pressed.

(4) If you try to use more than 7 disks, the largest ones will overlap each other when they are on adjacent poles. Make the width of a disk depend on the number of disks, so the widest disk is always 80 screen dots wide. Similarly, make the height of the disk depend on the number of disks, so that a full stack of disks is as high as the white rectangle on the screen.

(5) When the game is running and the user presses a mouse button, print (in the transcript) which disks are on each of the three poles.

(6a) Use a Form instead of a Rectangle for the shape of a disk in class HanoiDisk. Color the disk gray and give it a black border that is two screen dots wide. Class Form is in the category **Graphics-Display Objects.**

(6b) Use the followrwhile: message in class Form to give the disks smooth movement on the screen. The result should be nice-looking disks and smooth animated movement. Make the disks go in straight lines between their locations, or up and over, or in parabolas.

(7) There is a bug in classes HanoiDisk and HanoiDiskRules. If you create two instances of the game, there will be a conflict in setting the value of TheTowers. TheTowers is shared by all instances of HanoiDisk, when it should only be shared by all instances in a single game. Fix this by giving HanoiDisk a new instance variable that performs the same function as TheTowers. If you have done Problem 4, or just in case you will do it later. Thickness will no longer be a constant, and has the same problem. For completeness, turn every class variable (in class HanoiDisk) that is not a constant into an instance variable.

APPENDIX 5

HINTS FOR THE EXERCISES

The answers can be found on the following pages, but don't peek until you have tried using these hints.

(1) The act of drawing the rectangle is controlled in the method invert in class HanoiDisk. The code says:

```
invert
    Display reverse: rectangle
```

The variable rectangle is a simple Rectangle and does not actually have screen bits stored inside it. BitBit, Smalltalk's universal bit-slinging algorithm, performs several different types of operations (rules), and each goes through a mask to decide what bits to operate on. Base your changes to invert on the definition of reverse:. Find it by using the **messages** command in the middle-button menu of area D of the browser. (Find the code for invert in the browser, then move to area D and hold down the middle button.) The current mask is Form black, which means the whole rectangle. Form gray is also available.

(2) Modify the method moveUpon: in class HanoiDisk. The two new stopping points are (rectangle center x @ 120) and (destination center x @ 120). Split the delay up into three equal parts, one for each place the disk shows on the screen.

(3) You can read the mouse buttons by sending messages to Sensor, an instance of class InputSensor which is found in the category **System-Support.** Adding Sensor wartNoButton to the program will cause it to pause unless (or until) all buttons are up. You might want to look at the other messages in InputSensor to see what else you can do with the mouse.

(4) The width of a disk is controlled by the constant 14 in the next to last line of the method width:pole: in class HanoiDisk. Create a new class variable to hold the width increment, and compute the proper

value for it in whichTowers:. When the program runs with N disks, the largest disk has a width of N times the increment and the smallest is 1 times the increment wide. To make the height of a disk depend on the number of disks, make Thickness in whichTowers: be a function of the number of disks.

(5) As in Problem 3, add a line to moveUpon: in class HanoiDisk. The expression Sensor anyButtonPressed returns true if the user is holding a button down. The object that represents the whole game (TheTowers, an instance of AnimatedTowerOfHanoi) should be given the task of reporting the stacks, because an individual disk in the process of moving itself does not know what disks are on other poles. DeBne a new message in AnimatedTowerOfHanoi that prints the report in the Transcript.

(6a) A Form is a rectangle of bits that can be pasted on the screen. It knows its own extent (size), but not its location, so we still need the variable rectangle. Add an instance variable so that each disk can hold a Form. Create a Form by saying

```
Form extent: rectangle extent.
```

You can use the message fill:rule:mask: to paint bits into a Form. Look in the classes from which Form inherits its behavior to find the message displayOn:at:clippingBox:rule:mask:, and use it for displaying a Form on the screen.

(6b) The first argument to the message follow:while: should be a block of unevaluated code. It must deliver the next point where the upper left comer of the Form should be displayed. The second argument is another block that returns true until the disk reaches its destination. follow:while: assumes that the image of the disk is not on the screen when it starts to move it, and it does not leave the image on the screen at the end (so we have to compensate).

(7) After you have added an instance variable to the definition of HanoiDisk, you need to find all the places where the class variable you are replacing is used. An easy way to do this is to choose **class var refs** from the middle-button menu in area B. The system will ask you to frame a window, and it will list all of the methods that use the variable. You can see the code by clicking on the method name in the upper pane. Once you are looking at the code, you can modify it and **accept** it.

APPENDIX 6

ANSWERS TO THE EXERCISES

(1) Change the method for invert in class HanoiDisk to be

```
invert
  "Show a disk on the screen by masking an area and reversing it."
  Display fill: rectangle
    rule: Form reverse
    mask: Form gray.
```

The rectangle is still merged onto the screen using "exclusive or," but this time not all of the bits are changed. Only where the mask is black are bits on the screen reversed. We could have changed this code inside the reverse: method in class DisplayMedium, but since it is used by many parts of the system, all sorts of things (like highlighting in menus) would suddenly behave differently.

Notice that the modification we have made works for both AnimatedTowerOfHanoi and TowerByRules. The disks used by TowerByRules are instances of class HanoiDiskRules and they inherit the methods for displaying themselves from HanoiDisk.

(2) Change the method for moveUpon: in class HanoiDisk to be

```
moveUpon: destination
  "This disk just moved. Record the new pole and tell the user."
  pole <— destination pole.
  self invert.
  "straight up"
  rectangle center: (rectangle center x @ 120).
  self invert.
  (Delay forMilliseconds: 100) wait.
```

```
self invert.
"sideways"
rectangle center: (destination center x @ 120).
self invert.
(Delay forMilliseconds: 100) wait.
self invert.
"straight down to final location"
rectangle center: destination center - (0 @ (Thickness + DiskGap)).
self invert.
(Delay forMilliseconds: 100) wait.
```

(3) When Sensor is sent the message waitNoButton while a mouse button is pressed, it waits until you let go of the button. Insert this line:

```
Sensor waitNoButton.      "wait if button mouse is being held"
```

between any two statements in moveUpon: in HanoiDisk.

(4) Let's make a new variable to hold the difference in width between successive disks. Call it WidthDelta and make it shared by all instances of class HanoiDisk.

First select HanoiDisk in area B of the browser. From the middle-button menu in area B, choose **definition.** Add the class variable WidthDelta, as shown:

```
Object subclass: #HanoiDisk
  instance VariableNames: 'name width pole rectangle'
  class VariableNames: 'Thickness TheTowers DiskGap WidthDeita'
  poolDictionaries:''
  category: 'Kernel-Objects'
```

When you choose **accept** from the middle-button menu in area E, the system determines that WidthDelta is a new class variable, and adds it.

To use WidthDefta, replace the number 14 in the next to last line of the method width:pole:.

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

The only hard part of this solution is deciding what values WidthDelta and Thickness should have. The incremental width is equal to 80 divided by the number of disks. The thickness of a disk is the height of the white rectangle (220) divided by the number of disks, minus the space between disks. **Here** is a completely new version of whichTowers in class HanoiDisk:

```
whichTowers: aTowerOfHanoi
   | number |
   "compute the class-wide constants for disks"
   TheTowers <- aTowerOfHanoi.
   number <- TheTowers howMany.
   WidthDelta <- 80 // number,   "the widest disk is 80"
   DiskGap <- 2.
   Thickness <- (220 // number) - DiskGap.   "divide the height up evenly"
```

You can add a little class to this solution by not letting the disks be too thick. The purpose of making the height vary with the number of disks is to keep the top of the stack on the screen when there are lots of disks. When there are only three or four disks, the disks are quite thick and they don't look as good. Changing the last line to

```
   Thickness <— ((220 // number) - DiskGap) min: 14.
         "divide the height up evenly, but not too big"
```

limits the thickness to a pleasing 14 screen dots.

(5) Add a new line of code at the end of moveUpon: in class HanoiDisk:

```
   Sensor anyButtonPressed ifTrue: [TheTowers report].
   "If the button is pressed, ask the whole game to print its state"
```

It is important to put this line at the end of the method because we want to make our report when the state of the disks on the stacks (from which the report will be generated) agrees with the picture on the screen. We pass the task of actually doing the reporting to The-Towers in the form of a new message. Now let's write the code for that new message in class AnimatedTowerOf Hanoi:

```
report
   "Show in the Transcript a written report of which disks are on
     which towers"
   | aStack |
   1 to: 3 do: [:index |
      aStack *- stacks at: index.
      Transcript cr.
      Transcript show: Tower number', index printString.
      aStack isEmpty ifTrue: [Transcript show:' has no disks']
         ifFalse: (
            Transcript show:' has disks'.
            aStack reverseDo: [:disk |
               Transcript nextPut: disk name.
               Transcript space]]].
   Transcript cr.
   Transcript endEntry.   "force it to show"
```

(6a) Add an instance variable called image to class HanoiDisk. Initialize it by adding these lines to the end ofwidth:pole:

```
size >= 1000 ifFalse: [ "a normal disk"
    image <- Form extent: rectangle extent,    "set its size"
    image fill: image boundingBox
        rule: Form over
        mask: Form gray.        "fill in the halftone"
    image borderWidth: 2].      "give it a border 2 dots wide"
```

Use image as a pattern and invert the bits on the screen where the pattern has black bits. Change the method for invert to be

```
invert
    "Show this disk on the screen by inverting the bits where the Form is
        black"
    image displayOn: Display
        at: rectangle origin
        clippingBox: Display boundingBox
        rule: Form reverse
        mask: Form black
```

(6b) This solution is for a straight-line path between the disk's starting and ending positions. We start with the code for moveUpon: as it appeared before you worked any of the other exercises. All we have to do is to send the message follow:while: to the disk's image, and insert this between the call on invert and the code for moving the rectangle. We also need to define and initialize the local variables that hold the amount to move at each step and the number of steps completed.

```
moveUpon: destination    | count endPoint increment |
    "This disk just moved. Record the new pole and tell the user."
    pole <- destination pole.
    "Find the increment to move in a straight line path in 16 small steps"
    count <- 0.
    endPoint <-destination center - (0@(thickness+DiskGap)).
    increment <- endPoint - rectangle center //16.
    "remove the old image"
    self invert.
    "Move along the path. First block is next point, second is end condition."
    image follow: [rectangle moveBy: increment, rectangle origin]
        while: [(count <- count + 1) <= 16].
    "final position"
    rectangle center: endPoint.
    "display at its final position"
    self invert.
```

You can make the disks travel any path you want by varying the code that supplies Points to follow:while:. Try parabolas or semicircles.

(7) Choose class HanoiDisk in area B, and use the menu item **definition** to get its definition into area E. Add instance variables theTowers, thickness, and widthDelta (not capitalized to distinguish them from the class variables). As mentioned in the hint, choose **class var refs** to get a little browser on the methods that use each of the class variables. In each method, replace the class variable with its corresponding new instance variable. After accepting each of these changes, we must make sure the new instance variables are assigned values in every HanoiDisk that is created. To do this, we need to modify setUpDisks in AnimatedTowerOfHanoi. Previously, whichTowers: was called just once in each game to initialize the class variables in HanoiDisk. Instead let's call it once for every disk that is created.

```
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"
   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 whichTowers: self. "Create a disk"
      disk width: size pole: 1.
      (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 [
      disk <- HanoiDisk new whichTowers: self. "Create a disk"
      mockDisks at: index put: (disk width: 1000 pole: index)].
```

Note that we removed the line in which whichTowers: used to appear. We also need to make the same modification to setUpDisks in TowerByRules. (It's not exactly the same modification—we are creating a new instance of HanoiDiskRules instead of a new instance of HanoiDisk.)

Now you can start one game, interrupt it, and start a second game with a different number of disks. The two games interfere with each other only by occupying the same space on the screen; they no longer try to use the same variables.

INDEX

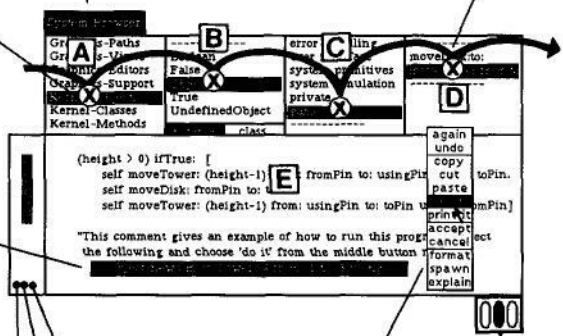
Entries in sans serif type refer to message (or procedure) names; entries in **Sans serif boldface** refer to menu commands.

J

A *window is active* when its title shows in reverse video. To use a window that is not active, move the cursor into any part of the window and click (press and release) the left button.

Select an item from a fixed menu by clicking on it with the left button.

Choose items left to right, general to specific.



To select text, press and hold the left button at the beginning of the passage, move to the end, and then release (the selected text will appea: in reverse video). New text typed in Area E (the cursor must be in Area E) replaces the selected passage.

Scroll bar Use the left button. The cursor changes shape as you move it from side to side within the scroll bar.

Click to bring the top line of the window down to the same line as the cursor.

Press the button and move up and down in the scroll bar to choose which part of the contents will show. The gray bar represents the fraction of the document that is currently visible.

Click to bring the line beside the cursor to the top of the window.

Pop-up menu Press and hold the middle button, move to the desired menu item, and release to choose it.

Mouse

The *left button* (red) selects text or fixed-menu items.

The *middle button* (yellow) controls pop-up menus for editing commands.

The *right button* (blue) controls pop-up menus for window commands.

A Taste of Smalltalk

Ted Kaehler
Dave Patterson

Seeing is deceiving.
It's eating that's believing.
— James Thurber

Written by two Smalltalk experts, this entertaining introduction to Smalltalk-80™ offers a brief tour of both the interactive programming environment and the language, for readers with some programming experience. Step-by-step instructions (accompanied by many pictures of the display screen) help the reader explore the unique user interface, while a series of example programs demonstrates the power of object-oriented programming.

Taking the Tower of Hanoi puzzle as their example, the authors compare a recursive Smalltalk program to similar programs in Pascal, C, and LISP, and then enhance their example with simple animation and a fully object-oriented algorithm. Observations on the nature of Smalltalk and advice on speaking "Smalltalkese" highlight the differences between Smalltalk and conventional programming environments.

A Taste of Smalltalk includes exercises (with hints and answers following) and a tear-out pocket reference card to aid the reader in exploring the system. The manuscript was tested extensively at Xerox's Palo Alto Research Center (PARC), where it was used to let new Smalltalk programmers "get their feet wet," and by students at the University of California at Berkeley.

TED KAEHLER was for eleven years a member of the Learning Research Group (later the Systems Concepts Group) at Xerox PARC, after receiving an M.S. in computer science from Carnegie-Mellon University. While at Xerox, he worked on virtual memory problems in Smalltalk (as well as other systems and language concerns), and created the explanation utility for the user interface. He is now in the Advanced Development Group at Apple Computer.

DAVE PATTERSON is professor of computer science at the University of California at Berkeley, where he led the design of a VLSI microprocessor for Smalltalk, called SOAR (Smalltalk on a RISC). In 1982, he received a Distinguished Teaching Award from the university.

Cover design by Linda Petterson.

Norton
W • W • NORTON & COMPANY NEW YORK • LONDON
ISBN 0-393-95505-2