TEK USERS MANUAL

Part No. 061-3440-00
Product Group 07

TEKTRONIX SMALLTALK

Please Check at the Rear of this Manual for NOTES and CHANGE INFORMATION

First Printing JUNE 1987

Tektronix
COMMITTED TO EXCELLENCE

Copyright © 1987, Tektronix, Inc. All rights reserved.

Tektronix products are covered by U.S. and foreign patents, issued and pending.

This document may not be copied in whole or in part, or otherwise reproduced except as specifically permitted under U.S. copyright law, without the prior written concent of Tektronix, Inc. P.O. Box 500, Beaverton, Oregon 97077.

Specifications subject to change.

TEKTRONIX ,TEK, and UTek are registered trademarks of Tektronix, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

PELLUCIDA is a trademark of Bigelow & Holmes.

HELVETICA and TIMES are registered trademarks of Linotype Corp.

UniFLEX is a registered trademark of Technical Systems Consultants, Inc.

Smalltalk-80 and XEROX are trademarks of Xerox Corporation.

MANUAL REVISION STATUS

PRODUCT: TEKTRONIX SMALLTALK USERS

This manual supports the following versions of this product: Image Version: TB2.2.1

REV DATE	DESCRIPTION
JUNE 1987	Original Issue

Table of Contents

Figures

Tables

Section 1
Introduction

INTRODUCTION

Tektronix Smalltalk is an enhanced version of the Smalltalk programming language and environment. It is based on Smalltalk-80[1] System Version 2. Smalltalk is a highly interactive, graphical, object-oriented system.

ABOUT THIS MANUAL

This manual is intended to serve the needs of new *and* experienced Smalltalk programmers. It includes information directed at both the new Smalltalk programmer and the experienced Smalltalk programmer. In general, the farther you go into the manual, the more experienced you should be in Smalltalk programming.

Prerequisites

The minimum requirements for a good understanding of the information in this manual are simply one or two years experience or schooling in some high level programming language such as Fortran, C, Pascal, or Lisp. Some exposure to operating system programming concepts, assembly language programming, or artificial intelligence programming concepts would also be helpful, but is not necessary.

Reading Path

To get the most benefit from this manual in the shortest time, follow the suggested reading path. In this manual, a "new Smalltalk programmer" is someone who fits the minimum requirements listed earlier. An "experienced Smalltalk programmer" is someone who has either studied the Addison-Wesley Smalltalk books listed later in this section, or someone who has had direct programming experience on a Smalltalk system.

For the New Smalltalk Programmer

As a new Smalltalk programmer, you should read sections 1, 2, and 3, in that order. Next study the Goldberg and Robson book, which is a reference book for the Smalltalk-80 system. After

1. Smalltalk-80 is a Trademark of Xerox Corporation.

this, you can go on to the *Tektronix Smalltalk Reference* manual and the other parts of this manual that interest you. Sections 4, 5, and 6 are intermediate in level and should probably be read before the remaining sections. However, be sure to go through the Model-View-Controller (MVC) tutorial in section 7 when you feel ready, since this important subject is barely described in the Addison-Wesley books.

For the Experienced Smalltalk Programmer

You should read through section 1 and the beginning of section 2, but you can skip section 3. However, be sure to read through sections 4 and 5 since they describe many features in Tektronix Smalltalk. You will be especially interested in section 7 about the Model-View-Controlerr paradigm since this important subject is barely described in the Addison-Wesley books. Read through the rest of the manual and the *Tektronix Smalltalk Reference* manual as your interests and needs dictate.

SUMMARY OF THE SECTIONS

Here is a summary of what you will find in this manual:

- Section 1 tells you what is in this manual and suggests a reading path through it. It tells you about the Addison-Wesley books on the Smalltalk-80 system and describes the Tektronix Smalltalk documentation.
- Section 2 is a brief introduction for those not acquainted with the Smalltalk-80 system. Section 2 presents the main features of the programming environment, language, and user interface in tutorial form.
- Section 3 is a brief conceptual overview of Smalltalk. Smalltalk language syntax and concepts are given along with an example of code. An introduction to programming tools such as the System Browser, System Workspace, File List, and Inspectors is presented. And, finally, a brief discussion of how programming is ordinarily done is presented.
- Section 4 describes many of the user-interface enhancements incorporated into Tektronix Smalltalk.
- Section 5 is intended for the advanced user and discusses programming tips, suggested programming style, and miscellaneous advanced programming topics. There is also quick introduction to the System Browser.
- Section 6 gives you detailed information about image and change management. This section should be read by programmers of all levels of expertise since it involves how to protect yourself from loss of work. Definitely read through this section before you begin work on application development.
- Section 7 is a detailed, code-oriented tutorial about the MVC programming paradigm. After completing this tutorial, you will be able to use the MVC paradigm in application programs of your own design.
- Section 8 describes in some detail the operating system interface of Tektronix Smalltalk. You should be familiar with Unix system programming, involving signals, processes, system calls, etc., to get the best understanding of this material.

- Section 9 describes Tektronix Smalltalk's implementation of fonts. There is a low-level introductory tutorial in addition to a task-oriented treatment for programmers with an intermediate level of expertise.
- Section 10 explains the characteristics of the Tektronix Smalltalk interpreter. Most programmers will not need to read through this section.
- Section 11 describes the operating system directories containing Smalltalk files. All programmers should read through this section. Be sure to take a look at the *fileIn* directory, where you will find code that you can incorporate into the system and experiment with.
- Appendices cover errors in the Addison-Wesley Smalltalk books, how Smalltalk treats key codes, and how to print out graphics images on a printer.

THE TEKTRONIX SMALLTALK DOCUMENTATION

In addition to this manual, the following documentation comes with Tektronix Smalltalk:

- Goldberg, Adele and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983. (Called in this manual the "Goldberg and Robson book," but known among Smalltalk programmers as the "blue book".) The Goldberg and Robson book is a formal explanation and description of the Smalltalk-80 language.
- *Tektronix Smalltalk Reference* manual. This manual includes classes and methods that have been added to Tektronix Smalltalk over and above those classes and methods in the Smalltalk-80 System Version 2.

In addition to this documentation, you will also want to consult the user's manuals that have been shipped with your Tektronix computer system.

The documentation for the Tektronix UTek operating system will also be useful to you.

The Goldberg Book

This manual makes reference to the following Addison-Wesley book about Smalltalk:

- Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984. The Goldberg book is an extensive introduction to the Smalltalk-80 system.

When this book is mentioned in this manual, the book is referred to as the "Goldberg book". (Some Smalltalk programmers refer to this book as the "orange" book.)

If you like, you can order the Goldberg book through Tektronix. Use the following part number.

062-8859-00	Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

GETTING STARTED

If you are familiar with the Tektronix computer system that you will run Tektronix Smalltalk on, turn now to Section 2. It will show you how to start up and run Smalltalk. (This manual assumes that you have made yourself familiar with the operation of your Tektronix computer system by reading through its user´s manual.)

Section 2
A Smalltalk Tutorial

OVERVIEW

This section is intended for the first-time user of the Tektronix Smalltalk system. In this tutorial, you will:

- Enter the Smalltalk system.
- Learn the commonly used features of the user interface.
- Run some example code.
- Learn how to save your work and exit back to the operating system.

This information is presented as a tutorial. Turn on your machine and perform the steps as you read this section.

The Smalltalk system overlays the hardware and operating system. This tutorial assumes that you are sufficiently familiar with these components to perform certain basic tasks. These are:

- Turn on your machine.
- Log into your machine.
- Access specific files and directories.
- Move through the directory structure.

If you cannot do these things, start by reading through the user manual that came with your machine, and come back to this tutorial when you are ready.

TUTORIAL

This tutorial assumes that you are performing the steps on your machine in the order that they are given. As you go through the tutorial and feel more confident about how the mouse, windows, and menus work, feel free to try operations similar to the ones described.

If, at any point during the tutorial, you accidentally perform an action which freezes the keyboard and mouse buttons, denying you access to Smalltalk, turn to the subsection entitled **Exiting Smalltalk**, later in this section. It explains how to get out of Smalltalk in the event of an unrecoverable error. If you exit Smalltalk in this manner, you will lose all work you have done since your image was last saved, unless you go through a special recovery procedure. However, you will be back at the operating system level and able to access your machine, and start Smalltalk, once again.

Entering the Smalltalk System

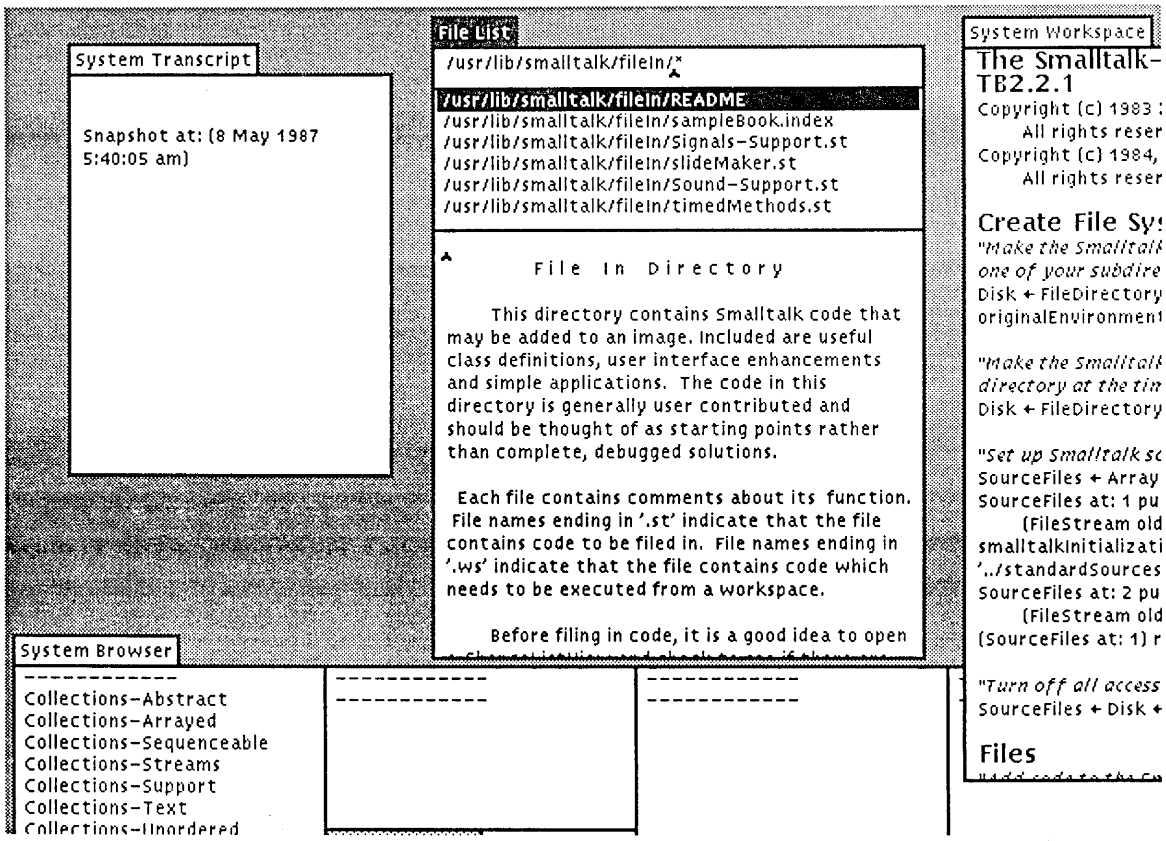
1. At the shell prompt, enter: *smalltalk*

2. The Smalltalk system files are large, so it takes some time for them to load. You will see the Smalltalk interpreter sign-on message:
   Tektronix Smalltalk Interpreter
   Version 2.2

3. While you are waiting for the Smalltalk system to load, make sure that you have a clear and level space to the right or left of the display to move the mouse around. Set the mouse down on this space with the cord leading away from you. This orients the three mouse buttons away from you. This is the correct orientation for the mouse.

Learning Mouse Mechanics

1. You will now see the initial Smalltalk display on the screen. See Figure 2-1. There are four windows visible; each has text within it and a border around it.



Snapshot at: (8 May 1987
5:40:05 am)

/usr/lib/smalltalk/fileIn/*

/usr/lib/smalltalk/fileIn/README
/usr/lib/smalltalk/fileIn/sampleBook.index
/usr/lib/smalltalk/fileIn/Signals-Support.st
/usr/lib/smalltalk/fileIn/slideMaker.st
/usr/lib/smalltalk/fileIn/Sound-Support.st
/usr/lib/smalltalk/fileIn/timedMethods.st

File In Directory

This directory contains Smalltalk code that may be added to an image. Included are useful class definitions, user interface enhancements and simple applications. The code in this directory is generally user contributed and should be thought of as starting points rather than complete, debugged solutions.

Each file contains comments about its function. File names ending in '.st' indicate that the file contains code to be filed in. File names ending in '.ws' indicate that the file contains code which needs to be executed from a workspace.

Before filing in code, it is a good idea to open

The Smalltalk-
TB2.2.1
Copyright (c) 1983 :
All rights reser
Copyright (c) 1984,
All rights reser

Create File Sy
"Make the Smalltal
one of your subdire
Disk ← FileDirectory
originalEnvironmen

"Make the Smalltal
directory at the tin
Disk ← FileDirectory

"Set up Smalltalk sc
SourceFiles ← Array
SourceFiles at: 1 pu
(FileStream old
smalltalkInitializati
'../standardSources
SourceFiles at: 2 pu
(FileStream old
(SourceFiles at: 1) r

"Turn off all access
SourceFiles ← Disk ←

Files

Collections-Abstract
Collections-Arrayed
Collections-Sequenceable
Collections-Streams
Collections-Support
Collections-Text
Collections-Unordered

3440-1

Figure 2-1. Initial Screen of Standard Image.

2. Grasp the mouse and move it back and forth and up and down around the space. The black arrow cursor moves in concert with your motions. The tip of the arrow is the part of the cursor that points to the item you wish to manipulate.

   Sometimes during your work in Smalltalk, the arrow cursor may change to a small square with a diagonal mark inside it. This is the garbage collection cursor. It indicates that the system is cleaning up and freeing available memory. While this cursor is visible, you will get no feedback from the keyboard or mouse buttons. This is normal system operation. Wait until the arrow cursor returns, and then continue as usual.

3. Move the cursor into one of the windows. The small title tab at the upper left corner of the window becomes highlighted. That is, the black letters turn white, and the white background turns black. You have just selected that window as the active window. Only one window is active at any time.

   Ordinarily, you can select a window to be the active window by clicking the left mouse button anywhere within it. The first window to be active after an image is brought up, however, is the first window into which the cursor is placed. When a window is active, you can access and manipulate its contents.

4. If you are using a 13" display, you cannot see the full 1376 by 1024 bitmap of the Smalltalk system at once. If you are using a 19" display, you can. If you are using the larger display, go on to the next section. If you are using a smaller display, however, you can access the full Smalltalk bitmap in the following manner. Move the mouse so that the arrow cursor moves up against the right side of the display. Watch the whole display move as you continue to move the mouse in the same direction. While doing this, you may reach the end of the cleared space. Just pick up the mouse and move it back to the other end of the space and continue the movement in the same direction. This is called *panning*.

5. Now move the mouse so that the arrow cursor moves down against the bottom of the display. Watch the whole display move as you continue to move the mouse. Again pick up the mouse and replace it on the cleared space as you need to, to continue the movement.

6. The joydisk in the upper lefthand corner of the keyboard is another way to pan the display. Push the joydisk down in each of the four directions, and watch the display move. What you are observing is a 640 by 480 hardware window accessing the Smalltalk system's 1376 by 1024 bitmapped display.

7. If you lose the cursor after this process, press function key <F12>. This immediately locates the cursor in the center of the visible screen.

Using the Mouse Buttons

1. The three buttons on the mouse give you access to many of the Smalltalk system's functions. They are used in the following manner:

   - The left button is used to select text, or to choose which window is to be active.
   - The middle button is used to pop up a menu to manipulate the contents of a window.
   - The right button is used to pop up a menu to manipulate a window itself.

2. You may either press a mouse button, or click it. In this tutorial, *pressing* a mouse button means to hold the mouse button down while you move the mouse to an appropriate position, and then release it. *Clicking* a mouse button means to press and immediately release a button.

Selecting Objects

1. Move the arrow cursor to the inside of the System Transcript window. (The System Transcript window is the window whose title tab is labelled System Transcript.) Click the left mouse button. The window's title tab turns to black, showing you that the window is selected and active. Left button activity anywhere in a window, including its title tab, activates that window. If any text in the window is highlighted, it is probably because you moved the mouse slightly while the left button was held down. Click the left mouse button again carefully, without moving the mouse, to unhighlight the text.

   In addition to some text, you will see a small dark caret cursor. The caret cursor is separate from the arrow cursor. The arrow cursor is used to move the caret, which indicates the point at which characters you type appear in the text.

2. Type a few characters. Take up the mouse again and move the arrow cursor between two characters along a line of text. Click the left mouse button. The caret cursor moves to the new point. Type a few more characters. Move the caret to the end of the paragraph, type a few carriage returns and then a sentence or two. The text automatically wraps within the window.

3. Move the caret cursor to the beginning of the text. Now press and hold down the left mouse button while you move the arrow cursor to the right and down in the System Transcript window. Release the button. Parts of the text become highlighted. Move the arrow cursor to the beginning of a line of text, press and hold down the left mouse button again, and sweep the cursor to the end of the line, and release the button. This highlights the line. Do this several times to feel how highlighting works.

   Now go to the very beginning of all text in the window. Click the left button twice. (If the window was not active before you tried this, you will need to click the left button three times.) The entire text is now highlighted. (If it looks like you have lost some text from the bottom or the top of the window, go on to the next section, and you will learn how to recover it.)

   When text is highlighted, it is selected. *Selection* means that you can manipulate the text in various ways.

4. Unselect all of the text in the System Transcript window, by clicking the left button again, anywhere in the window. Select a word in the window. Now type another word. Your word replaces the previously highlighted word. This is one way to replace text. Typing always inserts new text and by replacing the current text selection. This is true even if the current text selection is empty, as shown by the presence of the caret cursor.

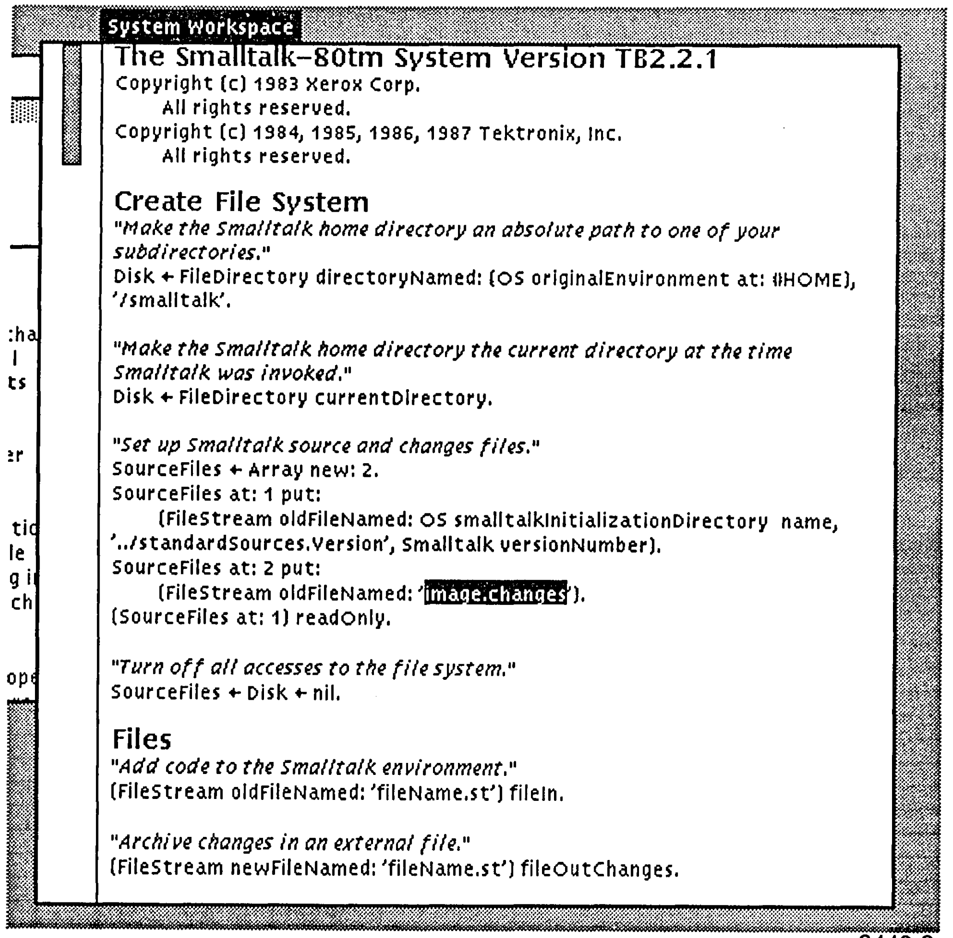
Scrolling Text in a Window

1. Activate the System Workspace window. See Figure 2-2. Move the arrow cursor directly to the thin rectangle at the left of the window. This is the scroll bar. In windows that have a lot of text, you can use this feature to move the text up or down, making new portions visible. There are three ways to use the scroll bar:

   - You can move text up one or more lines at a time.
   - You can move text down one or more lines at a time.
   - You can move quickly to a specific part.

There is a gray bar within the scroll bar; this is the scroll marker. The scroll marker's length, compared to the whole scroll bar, shows you approximately what proportion of the entire text is currently in the window. The scroll marker also shows you, by its position in the scroll bar, whether you are at the top, middle, or bottom of the entire text.

For example, if you see the scroll marker relatively small at the top of the scroll bar, you are looking at the very beginning of a lot of text. If the scroll marker fills the length of the scroll bar, all text associated with the window is currently visible in it.



```
The Smalltalk-80tm System Version TB2.2.1
Copyright (c) 1983 Xerox Corp.
     All rights reserved.
Copyright (c) 1984, 1985, 1986, 1987 Tektronix, Inc.
     All rights reserved.

Create File System
"Make the Smalltalk home directory an absolute path to one of your
subdirectories."
Disk ← FileDirectory directoryNamed: (OS originalEnvironment at: #HOME),
'/smalltalk'.

"Make the Smalltalk home directory the current directory at the time
Smalltalk was invoked."
Disk ← FileDirectory currentDirectory.

"Set up Smalltalk source and changes files."
SourceFiles ← Array new: 2.
SourceFiles at: 1 put:
     (FileStream oldFileNamed: OS smalltalkInitializationDirectory name,
'../standardSources.Version', Smalltalk versionNumber).
SourceFiles at: 2 put:
     (FileStream oldFileNamed: 'image.changes').
(SourceFiles at: 1) readOnly.

"Turn off all accesses to the file system."
SourceFiles ← Disk ← nil.

Files
"Add code to the Smalltalk environment."
(FileStream oldFileNamed: 'fileName.st') fileIn.

"Archive changes in an external file."
(FileStream newFileNamed: 'fileName.st') fileOutChanges.
```

Figure 2-2. System Workspace Window.

2. Slowly move the arrow cursor from right to left across the scroll bar. Three new cursors appear:

   - An upward-pointing half-arrow.

- A rightward-pointing arrow.
- A downward-pointing half-arrow.

Move the cursor back until you find the upward-pointing half-arrow. Press and release the left mouse button. The text moves down one or more lines for each press and release, depending on whether you are at the top, middle, or bottom of the scroll bar. The farther down the cursor is in the scroll bar, the greater the number of lines you scroll at a time. The upward-pointing half-arrow works analogously, in the downward direction.

Now place the cursor inside the scroll marker to access the rightward-pointing arrow. Press and hold the left mouse button, and move the mouse up and down. Observe that the scroll marker follows your motion. Release the left mouse button with the scroll marker positioned at the bottom of its bar. You are now looking at the end of the text in the window.

Move the rightward-pointing arrow to the top of the bar and click the left mouse button. The scroll marker follows the cursor. You are now looking at the beginning of the text in the window. Play with the scroll bar cursors until you feel comfortable with their operation. You may wish to activate other windows and experiment with the operation of their scroll bars as well.

Opening a Workspace Window

At this point, you have learned the basic operation of the scroll bars, mouse and left mouse button. You can now explore the middle and right mouse buttons further, and learn something about workspaces at the same time.

Workspaces are scratch pads where you can try out ideas and develop Smalltalk code.

1. Move the cursor against the gray background, access the middle mouse button pop-up menu and select workspace. A new cursor appears; it looks like a right angle. It marks the upper left corner of a new window.
2. Without holding any mouse buttons down, drag this new window anywhere over the display you like, even over already existing windows. When you have the new window where you want it, press and hold the left mouse button.
3. The right angle cursor now marks the lower right corner. Stretch the window out by moving diagonally away from the upper left corner. When the window is the size you wish to work with, release the left mouse button. Pressing the left mouse button anchors the upper left corner of the window, and releasing it anchors the lower right corner, fixing its size.

Evaluating Code in a Workspace

1. Activate the workspace window.
2. Type these Smalltalk expressions in the workspace:

```
3 + 4
0 sin
1 cos
100 factorial
Pen example
```

3. First, select the expression 3 + 4.
4. Press the middle mouse button down and hold it, while you scan the pop-up menu for the item print it. Roll the mouse down until this item is highlighted. Release the mouse button. You have caused your new line of Smalltalk code to be executed, and the result to be printed on the display. Every time you execute print it, the Smalltalk interpreter executes the selected code, prints the result on the display, and the action is recorded in your changes file.

   If your fingers slipped and you executed a different menu item instead, you can fix your mistake right now. Access the middle mouse button menu again and highlight the item undo. Then release the middle mouse button. This item undoes your previous action in the activated window. Now select the line and try the previous step again.

   If you access a menu, and then decide that you do not wish to execute any item on it, continue pressing the mouse button. Slide the cursor off the menu so that no item is highlighted. Now release the mouse button. The menu goes away when the mouse button is released, and because none of the items were highlighted at that time, none are executed.

   When you chose the middle mouse button menu command print it, the numeral 7 appeared just to the right of the expression. When you wish to evaluate a Smalltalk expression and see the answer, choose print it.
5. The answer to a print it command is already selected when it appears. This makes it easy to clean up your workspace. Choose the command cut from the middle button pop-up menu, and the highlighted expression 7 goes away.
6. Now select first 0 sin and then print it. Cut the result.
7. Follow the same sequence with 1 cos.
8. Follow the same sequence with 100 factorial. It may take a few moments for the system to compute this. When it does, you will see the answer has been computed to full 158-digit accuracy.
9. The last expression, Pen example, executes some example code which is a part of the Smalltalk system. Select it, but this time choose the do it command from the middle mouse button menu. The do it command executes the code, without printing the result on your display.
10. A square spiral shape appears on the display. As the method executes, the arrow cursor changes to an arrow cursor with a star. This cursor appears when Smalltalk is executing code. If you do not immediately see the spiral, pan the display to the right and down until you do.

    Choose do it when you want to execute selected Smalltalk code. Choose print it when you want to see the result of executing selected Smalltalk code.
11. When the display gets cluttered, as it is after drawing the square spiral, you can restore it with a new middle mouse button pop-up menu. To access this menu, position the cursor outside all windows, against the gray background. Move the cursor to the gray background

and press and hold the middle mouse button. Go to the top of the menu and select restore display. The entire display will be redrawn. The square spiral disappears. It was not saved, so it is gone.

Later, you will be able to examine the code that is executed when you execute or print these expressions.

Manipulating Text in a Workspace

1. To facilitate common text manipulation tasks, the Smalltalk system has implemented the commands undo, copy, cut, and paste in the middle mouse button pop-up menu in a workspace. Highlight the expression Pen example in the workspace. Select the middle mouse button command cut. It removes the highlighted text. It also puts the text into a temporary buffer.
2. Now choose the middle mouse button command undo. The expression reappears.
3. Highlight Pen example as before. Now select copy from the middle mouse button menu. You have just copied the text Pen example into the buffer.
4. Now move the arrow cursor into the gray background and press and hold the middle mouse button. Choose the workspace command and open a new workspace window. With the arrow cursor in the workspace window, choose paste from the middle mouse button menu. The text Pen example appears in the new workspace. You can copy and paste virtually any size block of text you require. copy and paste allow you to shift blocks of text between windows.
5. If you would like to get rid of the workspace window after you are done with it, press the right mouse button and choose the command close. A notifier window pops up, asking you for confirmation. When you close a workspace, you lose all the text in that workspace. In this case, though, click the left mouse button on the yes pane on the left.

   You must respond to a notifier window by clicking on either yes or no. If you move the cursor away without choosing either one, the window flashes until the cursor returns. This feature warns you when an action you are trying to perform would result in losing work.

Accessing Smalltalk Code

The System Browser gives you access to all code in the Smalltalk system. With it, you can copy and modify the code as you require.

The System Browser has five major panes: four small ones at the top and one large one below. From left to right, the top panes are:

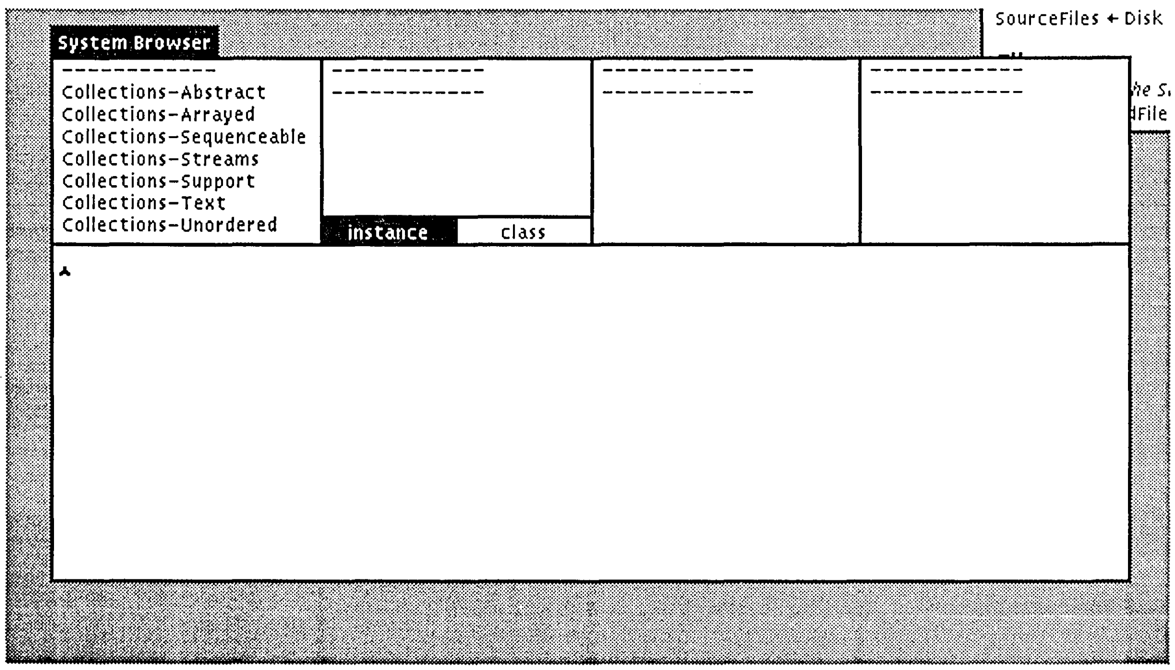
- The class category pane.
- The class pane.
- The message protocol pane.
- The message selector pane.

The large pane below is called the text pane.

The class category and message protocol panes (the first and third panes) are aids to organization. They do not have any counterparts in Smalltalk code. They are there to help people remember where to find a particular class or message.

The class and message selector panes access Smalltalk code.

1. Activate the System Browser. See Figure 2-3.
2. Move the arrow cursor into the class category pane. Click the left mouse button on the class category Numeric-Numbers. (Use the scroll bars if the category is not visible to you.) If the category is selected, it will be highlighted.
3. The class pane now has a list of Smalltalk class names in it. The bottom pane now contains Smalltalk template code. This is code which can be edited and then executed.



3440-3

Figure 2-3. System Browser Window.

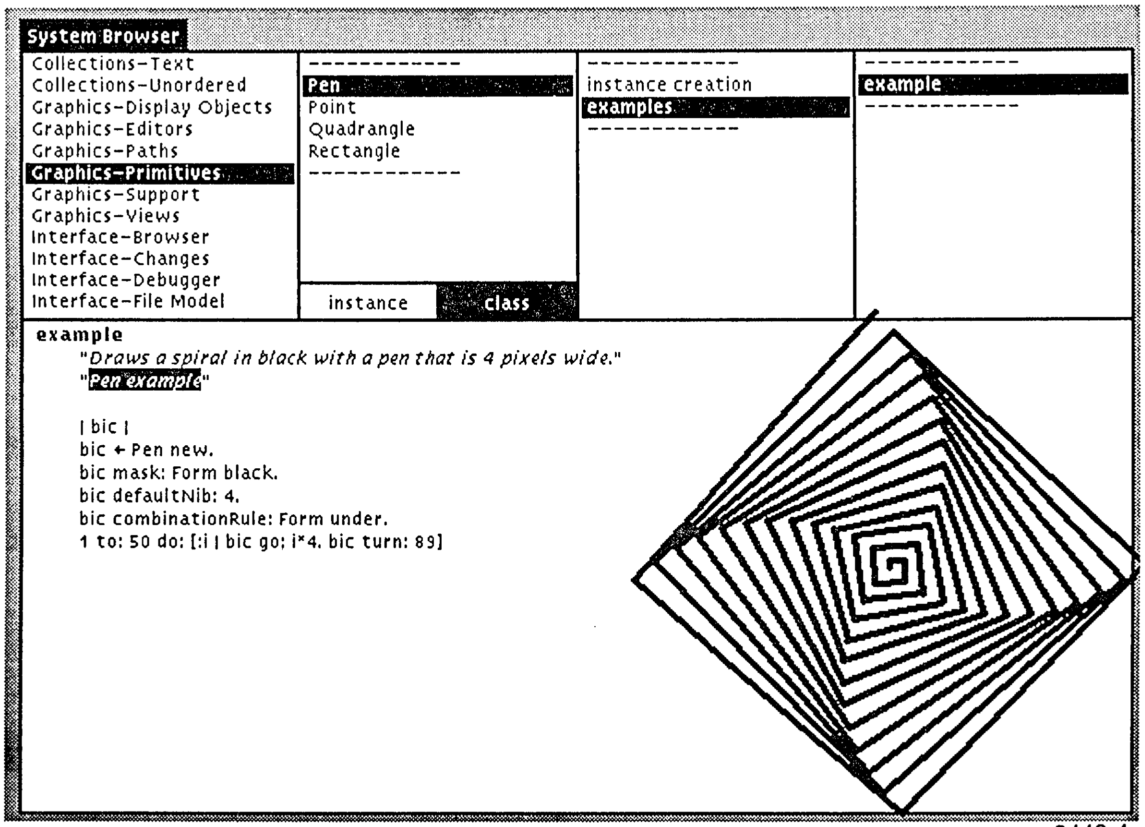
4. Move the arrow cursor to the class pane. This pane has two boxes at the bottom containing the words instance and class; select instance by clicking with the left button. Now select the class name Integer.
5. Now the message protocol pane fills with a list of message protocols, and the bottom pane template code changes. Select factorization and divisibility in the message protocol pane. Again the next pane over, the message selector pane, fills with a list of expressions. (If your System Browser is small, you will see only the leftmost portion of the words.)
6. The expressions in the last pane are actual Smalltalk code message selectors. These expressions are the parts of the Smalltalk code that tell objects to perform operations. In

the rightmost pane, select factorial. This message selector tells an object representing an integer to perform the factorial operation upon itself.

7. In the text pane below, you now see the actual Smalltalk code that is executed when you compile and run a Smalltalk expression such as 100 factorial, as you did previously in the workspace. Use the scroll bars if necessary to examine the code.
8. After you have examined the code to your satisfaction, go back to the message selector pane and select a different message selector. You can then examine the code used to perform that operation.
9. At this point, if you like, you can go back to the message category pane, select a different message category, and then, select any new message selector you see in the message selector pane.
10. Go back to the leftmost pane and choose a different class category. Start the process all over again, and explore the Smalltalk code for a different class. In this way, the System Browser allows you to access and learn about all code in the system.

Executing Example Code

1. Use the System Browser to find and execute the Pen example code. Go to the class category pane (the leftmost one) and select the category Graphics-Primitives. (You may have to use the scroll bar to find the category in the list.)
2. Next select the class Pen. Also select the class box in the bottom of the pane.
3. Then select examples in the message categories pane.
4. Then select example in the message selectors pane. In the text pane you will see the example method for the class Pen. See Figure 2-4.



```
example
	"Draws a spiral in black with a pen that is 4 pixels wide."
	"Pen example"

	| bic |
	bic ← Pen new.
	bic mask: Form black.
	bic defaultNib: 4.
	bic combinationRule: Form under.
	1 to: 50 do: [:i | bic go: i*4. bic turn: 89]
```

Figure 2-4. Pen Example Code in System Browser Window.

5. To run this example of Smalltalk code, look at the end of the code and find the expression "Pen example". The expression Pen example may look familiar to you. It is the same expression you ran in the workspace, but it has been enclosed in double quotes within the method to turn it into a comment.

6. Select just the Smalltalk expression Pen example, without the quotes. There are two ways to do this:

   - Put the arrow cursor point between the opening quote and the first letter. Click the left mouse button twice without moving the mouse. This is called *double-clicking*.

   - Put the arrow cursor point between the opening quote and the first letter. Hold the left mouse button down while you run the cursor to the right until you reach the last letter. Then release the button. This is the technique of selection you learned previously.

   You can use the second technique to select an item or items. Double-clicking works on any expression within delimiters, such as quotation marks or parentheses. For double-clicking to work, you must place the arrow cursor point immediately after the initial delimiter, if there are any spaces between it and the first character on the line.

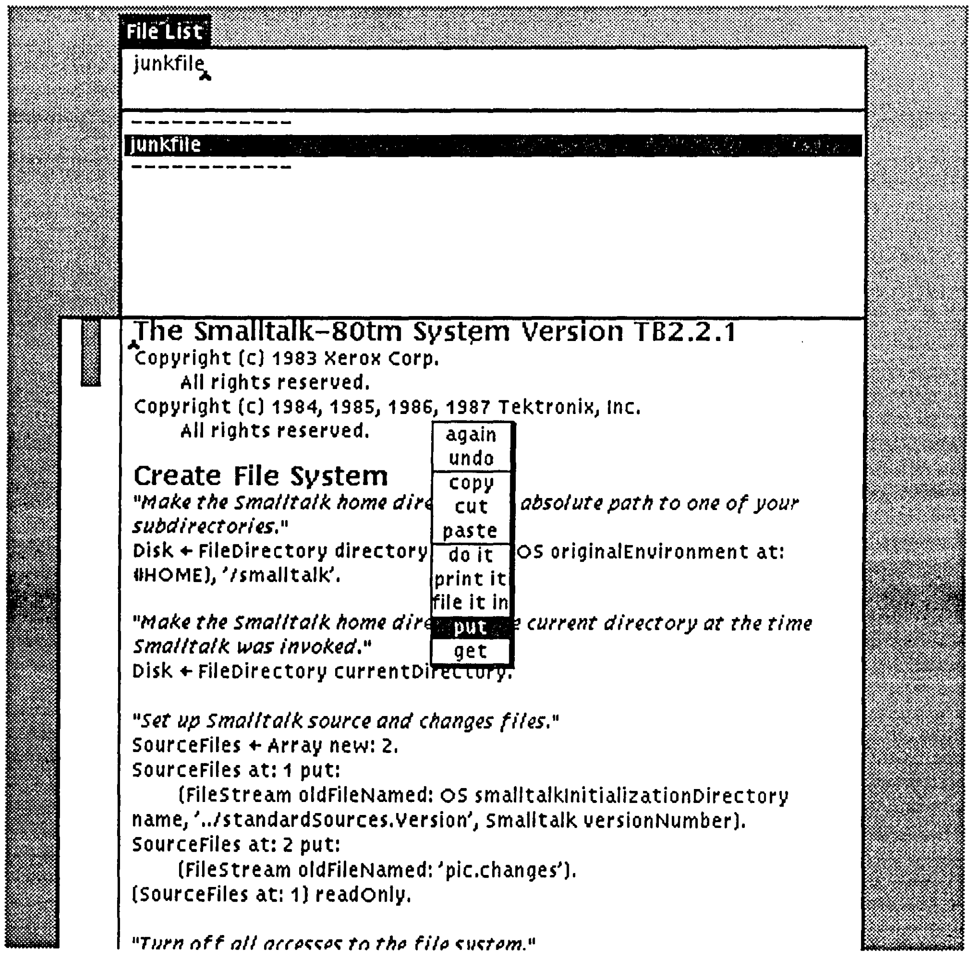
7. With just Pen example highlighted, press and hold the middle mouse button. A pop-up menu appears. Execute the command do it. The Smalltalk system understands the expression and executes it just as it did in the workspace.
8. The square spiral shape appears on the display again.
9. Restore the display.

Altering the Code

1. Examine the Pen example method code. You can pick out certain constants that affect certain aspects of the figure. Use the left mouse button to select the 4 in the expression bic defaultNib: 4. Now type 6. The expression changes to bic defaultNib: 6. Make sure you have changed just the 4 to a 6 and not the period that ends the expression.
2. Press the middle mouse button. Find the command accept in the pop-up menu. (It is below do it.) Select accept. The Smalltalk busy cursor appears for a few seconds. You have just recompiled the example method to enter the altered code into the Smalltalk system. You are now ready to do it and observe the result of your change.
3. Select Pen example again, and then do it. The lines of the figure are thicker than before.

Communicating with the Operating System

You can communicate with the operating system from within Smalltalk to look at the contents of a directory or files, write text in a window to a file, or get the contents of a file. To do this, use the File List window. See Figure 2-5.



```
junkfile

------------
junkfile
------------

The Smalltalk-80tm System Version TB2.2.1
Copyright (c) 1983 Xerox Corp.
     All rights reserved.
Copyright (c) 1984, 1985, 1986, 1987 Tektronix, Inc.
     All rights reserved.

Create File System
"Make the Smalltalk home dire          absolute path to one of your
subdirectories."
Disk ← FileDirectory directory          OS originalEnvironment at:
#HOME), '/smalltalk'.

"Make the Smalltalk home dire          current directory at the time
Smalltalk was invoked."
Disk ← FileDirectory currentDirectory.

"Set up Smalltalk source and changes files."
SourceFiles ← Array new: 2.
SourceFiles at: 1 put:
     (FileStream oldFileNamed: OS smalltalkInitializationDirectory
name, '../standardSources.Version', Smalltalk versionNumber).
SourceFiles at: 2 put:
     (FileStream oldFileNamed: 'pic.changes').
(SourceFiles at: 1) readOnly.

"Turn off all accesses to the file system."
```

3440-5

Figure 2-5. File List Window.

1. If you need to, restore the display.
2. If you want a bigger File List, choose the command frame from the right mouse button menu, and frame the window exactly as you did when you got a new workspace.
3. The File List window has three panes.
   - The top pane accepts file or directory names for you to access. You can use wildcard characters to access all names that match a specific pattern.
   - The middle pane lists the contents of a directory and allows you to choose a specific file to access.
   - The bottom pane gives you information about a file, including its contents.

At this point, the top pane contains the path */usr/lib/smalltalk/fileIn/**. The asterisk * is the Smalltalk wildcard character standing for all possible character combinations. You are therefore getting a list of all the files in that directory.

The middle pane contains the list of the files in that directory. The first file name, */usr/lib/smalltalk/fileIn/README*, is selected.

The contents of the README file is displayed in the bottom pane. Take a moment to read it now, if you wish.

4. In the list of files, you see a file named Animation.st. Click the left button on it to select it. In the bottom pane, you see a message about the size of the file and the time it was created. It looks something like this:

```
8140 bytes
18 March 1986
6:18:03 pm
```

5. From the list (middle) pane, execute get contents from the middle button menu. The bottom pane fills with the information in the file. There is a short message at the beginning, explaining how to use the file. Read it now.
6. Now execute file in from the same pop-up menu. This reads the contents of the file into your image, so the code is available for you to use.
7. Look near the top left of your screen, and you see the animation executing. Move the cursor around various parts of your screen. Move it into and out of the animation area. Move it next to the transparent box.
8. When you are finished with the animation, click any mouse button while the cursor is in view. The animation stops. Restore the display.
9. Select the line reading WindowNode example from the middle pane of the FileList window, and execute do it, to execute the animation again. Because the code has been filed into the system, it is now available to you by executing this expression from any window or workspace.
10. Now move the cursor into the top pane of the File List window and activate the window by clicking the left mouse button within it. Click the left mouse button twice more within the top pane, at the beginning of the pathname, and you will select the entire line.
11. To replace the text in the top pane, type */usr/lib/smalltalk/**. Or instead, simply select the portion that reads *fileIn*, and cut it. Then press the middle mouse button and choose accept form the menu. The middle pane of the window fills with the names of files in the directory you have chosen to list. Select several items and look at the information in the bottom pane. Some of the items are directories.
12. Choose the name system; this is a directory.
13. Choose the middle mouse button menu command list contents. The bottom pane now contains a list of the files in that directory.

14. From the middle mouse button menu, choose the command spawn. A second File List window opens. Frame it as you framed the new workspace. This File List window is identical to the one you have just been working with, and can be manipulated in the same way. When you are done with it, choose the right mouse button menu command close and go back to the original File List window.

Writing Files Out to the Operating System

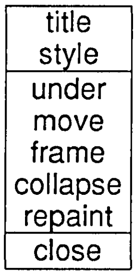
Suppose that you would like to create a new file and put some text from the Smalltalk system into it.

1. In the top pane of the File List window, select the entire line and replace it by typing the name of the new file. Type *junkfile*. Then select the middle mouse button menu command accept. You should see *junkfile* appear in the middle pane.
2. Now select *junkfile* in the middle pane with the left mouse button. The message — new file or directory — appears in the bottom pane. (If it does not, try another file name until you can be sure you have a truly new file name.)
3. Select the — new file or directory — message and cut it from the window. Now activate the System Workspace window and select all of the text in that window. You can do this by going to the very beginning of the text and clicking the left mouse button twice. Select the copy command from the middle mouse button menu.
4. Go back to the bottom pane of the File List window and activate it. Select the paste command from the middle mouse button menu. The entire text from the System Workspace window is now in the File List window. Unselect it by clicking the left mouse button once. Then use the scroll bar to verify that this occurred.
5. Still in the bottom pane of the File List window, press the middle mouse button and look near the bottom of the pop-up menu. Choose the command put. It takes a short time for the Smalltalk system to write the contents of the File List window into the file called junkfile. While it is doing so, you will see the cursor change to the writing pencil cursor.
6. To verify this, go to the middle pane and unselect junkfile by clicking on it. Now select junkfile again. The bottom pane of the File List window now contains information about the size of the file.
7. To inspect the contents of the file, go to the middle pane and use the middle mouse button to select the command get contents. This also takes a short time. While Smalltalk is getting the contents, you will see the cursor change to the reading spectacles cursor. The contents of junkfile reappears in the bottom pane of the File List window. Font information is not preserved after the text has been written out and read back into the system.

Manipulating Windows

The right mouse button menu contains commands to manipulate windows, instead of the contents of windows.

1. Choose a window on the display and activate it by clicking with the left mouse button. Now, press and hold the right mouse button. You will see this menu:



- title allows you to change the title of a window.
- style allows you to choose a text style for your window. This may take some time. (See Section 7, **Fonts in Smalltalk,** for further discussion of text styles.)
- under allows you to bring a window into view if it is under another window. (Windows overlapping each other are arranged in an internal stack. under pops the window on the bottom of the stack to the top. Your cursor must be directly over the hidden window, however.)
- move allows you to change the position of a window on the display.
- frame allows you to change the size and position of a window.
- collapse allows you to delete from the screen all of a window except its title tab, which you can then move.
- repaint allows you to redisplay the contents of the current window.
- close allows you to remove a window entirely from the display. This discards all of the unsaved contents of the window.

Experiment with each of these commands until you feel reasonably comfortable with their operation.

Saving Your Image

Saving your work in Smalltalk is called *making a snapshot,* or *saving your image.* It is called making a snapshot because, when you next invoke your image, your screen appears exactly the way it appeared when you took the snapshot, except for any unscheduled graphics, such as the results of the Pen example, which you may have been running.

1. To save your work, move the cursor to the gray background.
2. Choose the middle mouse button command save.
3. A window pops up with the default file name, image. This is called a prompter window. You can accept the default name by entering a carriage return. Or you can enter your own file name.

If you change your mind and do not wish to save your work at this point, cut the text in the prompter window and simply enter a carriage return. The prompter disappears and you can continue working.

4. Save the image file as image, or enter your own image file name. The system now makes the snapshot of your current image. During the process of writing your image file, you will see the hourglass cursor. This cursor means the system is busy. You may also see the garbage collection cursor again. When the system has finished saving the image file, you will see the arrow cursor again. This means the system is ready for input.

It is a good idea to take a snapshot of your image regularly throughout the day, before breaks or after you have accomplished an important task. That way, your work is backed up if the system fails.

Exiting Smalltalk

There are three ways to exit Smalltalk.

- You can save you work and then quit.
- You can quit without saving your work.
- You can interrupt the process in the event of a system failure.

1. To save your work and then quit, move the cursor to the gray background.
2. Choose the middle mouse button command save, then quit.
3. A window pops up with a default file name. This is called a prompter window. You can accept the default name by entering a carriage return. Or you can enter your own file name.
4. Save the image file as image, or enter your own image file name. The system now makes the snapshot of your current image. During the process of writing your image file, you will see the hourglass cursor. This cursor means the system is busy. You may also see the garbage collector cursor again. When the system has finished saving the image file, you will be back at the operating system level.

After you are through with this tutorial session, you may want to quit without saving your work.

1. To quit without saving your work, move the cursor to the gray background. Use the middle mouse button menu command quit. If you are ready to quit, do so now.
2. A confirmation window pops up, giving you the opportunity to change your mind and save your work. You have three choices. You can:
   - Quit without saving your work.
   - Save your work, then quit.
   - Continue working and neither save nor quit.

   Choose quit, without saving. You will be back at the operating system level.

If the system fails, or a serious bug in one of your programs corrupts the image, you may not be able to access the keyboard or any of the pop-up menus. If you can get no other input, you need the emergency exit from Smalltalk.

CAUTION

If you use the method described below to exit Smalltalk, you will lose all the work you have done since you last saved your image, and you will have to go through a special recovery procedure. This procedure is discussed in Section 6.

1. Press the keys <SHIFT>-<CTRL>-<BREAK> simultaneously. You have now accessed the terminal emulator.
2. Press the keys <CTRL>-<C> simultaneously. You are now back at the operating system level.
3. If you are working on a small screen, and do not see any change on it after pressing these keys, try panning all the way to the left and top of the virtual display. (Panning still works.) The operating system prompt is visible in a small portion of the display near the upper left.
4. Enter the operating system command *clear* to clear the display.
5. Enter the operating system command *conset default* to get rid of the Smalltalk cursor.

Invoking Your Image File

After you have completed your first Smalltalk session, two new files reside in the directory from which you invoked Smalltalk.

- Your image file holds the Smalltalk image that incorporates all the work you have saved. The default name is image, unless you chose to name it something else when you saved the file from within Smalltalk.
- The file image.changes holds the changes you have made to the standard image. This file is useful for recovering from a system or application failure. (See Section 6, **Change Management**, for further information on change management and crash recovery.)

When you wish to take up where you left off with your work, invoke your snapshot image file, *image* (or whatever name you used when you saved the file).

1. At the operating system level, change to the directory from which you invoked Smalltalk.
2. Then type *image* instead of *smalltalk* at the operating system prompt.

You have now finished this tutorial and are ready to go on to more advanced Smalltalk concepts. Congratulations!

Section 3
Smalltalk Concepts

OVERVIEW

This section consists of three main subsections.

The Smalltalk Language	Presents concepts basic to the Smalltalk programming language. It also explains Smalltalk language syntax. Finally, it gives an example of Smalltalk code, showing how these concepts and syntax are used to program in Smalltalk.
Programming Tools	Presents commonly used tools available to the programmer in Smalltalk. The System Browser, workspaces, the System Transcript, the System Workspace, the File List, inspectors, and debuggers are all discussed.
Extending the Language	Consists of two tutorials. The first teaches you how to add a new method to an existing class. The second teaches you how to add a new class to the system.

THE SMALLTALK LANGUAGE

The Smalltalk-80 system is an interactive programming environment. The system is designed so that the keyboard is used only to type text. Other actions are done using the mouse buttons and pop-up menus. The system is a visual one, providing immediate feedback and allowing you to work without having to remember commands.

Key Concepts

Everything in Smalltalk is an object. An *object* has some private memory called instance variables. It also has access to a set of operations called *methods.*

A *message* is a request for an object to carry out one of its operations. A message is sent to a *receiver.* The receiver of the message carries out the specified operation by performing a method associated with that message. A result is always returned. The default result is the receiver itself.

Messages are similar to procedure calls; they specify an operation to perform. Unlike procedure calls, however, the operation is specified indirectly by a message name. The name of a message is called a *selector.* The interpretation of a message selector is determined by the class of an object, rather than by a procedure name with a single interpretation. The same message selector may specify different methods in objects of different classes.

A *class* is a special kind of object. Each class contains a template for creating a specific kind of object. An individual object described by a class is called an *instance* of that class. The class specifies the messages that instances of that class respond to, and in what manner they respond.

Classes are arranged in a hierarchy. Classes are defined as *subclasses* of other classes. Methods defined for instances of a class are inherited by instances of its subclasses.

Syntax

This subsection gives the basic rules of Smalltalk syntax. See Goldberg and Robson, Chapter 18, for more information on this subject.

Objects and messages are referred to by alphanumeric identifiers.

A method is a sequence of expressions. There are four types of expressions:

- Identifiers (variables)
- Literals (constants)
- Messages
- Blocks

Identifiers

Identifiers identify regions of private memory reserved for the storage of instance variable values. Examples of identifiers are: foo, aDate, page23, Smalltalk. Their values depend on their context. They must begin with a letter: lowercase for local variables and instances of a class, uppercase for global variables and class names.

A *local* variable is known only to the method, block, or workspace within which it is used. A *global* variable is known throughout Smalltalk. Smalltalk is a global variable.

Five pseudovariables are used frequently:

- nil
- true
- false
- self
- super

Undefined objects are initialized to nil. true and false are Boolean objects. The values of self and super depend on their context. You cannot reassign the values of these pseudovariables.

Literals

Literals can be numbers, characters, strings, symbols, or arrays. Examples of numbers are: 3, -4, 2.5, 1e6, 8r23, 9/5. Fractions are true rational numbers. Instances of class Fraction have two instance variables: a numerator and a denominator.

Characters are prefixed with a $. $A means "the character 'uppercase A'." Other examples of characters are: $a, $@, $7, $$.

Strings consist of one or more characters, delimited by single quotes. Because single quotes are string delimiters, single quotes within strings must be doubled. Examples of strings are:
'abc 123' 'You can''t do that.'.

Symbols are prefaced with a #, except when embedded within an array. Examples of symbols are: #July, #at:put:, #Fraction.

An array is delimited by parentheses. It is also preceded with a #, except when embedded within other arrays. Examples of arrays are: #(1 2 3) #(June, July, August) #('yes' 'no' 'maybe').

Smalltalk has two special literals, the leftward arrow ← and the upward arrow ↑. A special key on the keyboard shows these symbols. The leftward arrow ← is the assignment operator. It assigns values to variables. For example:

```
aNum ← 2 + 3.
aString ← 'Hello, world!'.
aDate ← Date today.
```

The upward arrow ↑ returns the value of the expression. For example:

```
↑ $a asUppercase.
```

Messages

A message is implemented by a method. A method is composed of Smalltalk code. It is referred to by a message selector. The same message selector can be used by different classes. Objects of different classes may respond to a given message selector by executing different methods. Methods always return a value.

There are three kinds of message selectors. They are:

- Unary
- Binary
- Keyword

Unary messages take no arguments. Examples of unary messages are given below. Explanations of the examples are to the right.

2 sqrt	Two, compute your square root and return its value.
$A class	$A, what class are you an instance of?
Sensor waitButton	Sensor, answer the coordinates of the point where the mouse button is pressed.
#(a b c) size	Array, how big are you?
Rectangle fromUser	Rectangle, get your upper left and lower right corner coordinates from the user.

Binary messages take one argument. Examples of binary messages are given below. Explanations of the examples are to the right.

3+4	Send the object 3 the message + with the argument 4. This is equivalent to: 3, add 4 to yourself.
2*3	Send the object 2 the message * with the argument 3, equivalent to: 2, multiply yourself by 3.

500@225	The @ message creates points, when sent to a number. This line sends the object 500 the message @ with the argument 225. The receiver is the *x* coordinate, and the argument is the *y* coordinate. The general formula is *x @ y.*
'Four score and ','7 years ago'	The comma message is sent to a string with an argument of another string. The value returned is the concatenation of both strings.
a<=b	Send the object a the message less than or equals with the argument b. This message returns either true or false.

Keyword messages are more complex. They are composed of any number of words, each followed by a colon. A keyword message takes as many arguments as it has words. Examples of keyword messages are given below. In each example, the keyword message is italicized so you can see its components. Explanations of the examples follow.

anArray *at:*1 *put:*'Joe'
The keyword message at:put: puts the specified string,
'Joe', at the specified index 1.

aDate *newDay:* 15 *month:* #July *year:* 87
The keyword message newDay:month:year: makes an instance of
class Date.

In Smalltalk, all message selectors begin with a lowercase letter.

Blocks

Blocks are sets of deferred expressions somewhat like methods. They are delimited by square brackets. For example:

```
[index ← index + 1.
 anArray at: index put: 0]
```

Blocks can have block variables. A block variable exists during the execution of the block. Each block variable is preceded with a colon. It is separated from the first expression of the block by a vertical bar. For example:

```
summation ← [:anArray | total + anArray size]
```

Creating a block does not cause it to be executed. Blocks are executed as a whole when the block is evaluated. A block is evaluated when it is sent the message value or value:. For example:

```
ba ← [c ← 3 + 4]
```
is evaluated when the expression
```
ba value
```
is executed.

Because of this deferred evaluation, blocks may be used to select which of several sets of expressions get executed. For example:

```
anArray size = index
          ifTrue: [sum ← 0]
          ifFalse: [sum ← 1]
```

Blocks can be used when you want to repeat sets of expressions, either conditionally or unconditionally. For example:

```
sum ← 0.
index ← 1.
[index <= anArray size]
          whileTrue:
             [sum ← sum + (anArray at: index).
               index ← index + 1]


result ← 1.
7 timesRepeat:
          [result ← result + 2.718]
```

A block returns the value of the last expression evaluated. For example:

```
sum ← anArray size = index
            ifTrue: [0]
            ifFalse: [1]
```

The expression above returns zero if the first statement is true, and one if it is false.

Parsing Expressions

Unary messages are parsed from left to right. Binary messages are also parsed from left to right, unlike many other programming languages. In Smalltalk, x + y * z is equivalent to (x + y) * z. Unary messages take precedence over binary messages. That is, 1 + 2 sqrt is equivalent to 1 + (2 sqrt).

Keyword messages of more than one word, for example at:put:, are parsed as one message. Binary messages take precedence over keyword messages. That is, anArray at: 20 put: average + bias is the same as anArray at: 20 put: (average + bias), where average and bias are variable identifiers.

Expressions within parentheses are evaluated before expressions outside parentheses.

Spaces, tabs, and carriage returns may be used within an expression. A period separates expressions. The last expression in a block or method should not have a period. If an expression which returns a value is terminated with a period, a syntax error results.

The same object may be sent a sequence of messages. In that case, you need not repeat the object's name. The messages may instead be *cascaded*: that is, separated by semicolons. For example, the expression:

```
aPen down; turn: 90; go: 50; turn: 90; go: 150.
```

sends the same instance of class Pen, called aPen, five messages in a specific order. They are:

1. Set yourself down on the drawing surface.
2. Turn 90 degrees in a clockwise direction.
3. Go 50 pixels.
4. Turn 90 degrees again.
5. Go 150 pixels.

The following list summarizes these parsing rules.

- Unary messages take precedence over binary messages.
- Binary messages take precedence over keyword messages.
- Messages of the same kind are parsed left to right.
- Expressions within parentheses are evaluated before expressions outside parentheses.

An Example of Smalltalk Code

In the tutorial in Section 2 you saw the code for the Pen example. The code draws a spiral on the display. You may wish to examine this code more carefully now. First, here is the entire example.

```
example

"Draws a spiral in black with a pen that is 4 pixels wide."

"Pen example"

        | bic |
        bic ← Pen new.
        bic mask: Form black.
        bic defaultNib: 4.
        bic combinationRule: Form under.
        1 to: 50 do: [:i | bic go: i*4.  bic turn: 89]
```

Now each component of the method is explained individually.

example
This is the message selector for the method.

"Draws a spiral in black with a pen that is 4 pixels wide."
This is the comment. A comment line is included immediately following the message selector in

many of the methods found in the Smalltalk system. This initial comment summarizes the method. Comments are enclosed in double quotation marks and may be interspersed anywhere in the code. A blank line conventionally follows the initial comment.

"Pen example"
This is another comment. However, within the quotes, it is also a valid expression in Smalltalk. The message selector example selects this particular method. Thus, the code within the quotes says to send the example message to the class Pen. Pen looks for a message selector named example, finds it, and executes the code you are examining. A commented Smalltalk expression at the beginning or end of a method is frequently used to show how to execute the method.

| bic |
This line declares the temporary variables, enclosed between vertical bars. Each is separated by a space from the next one. Here bic is the instance of the class Pen performing the drawing, and is the only temporary variable. Temporary variables conventionally begin with a lowercase letter.

bic ← Pen new.
This line assigns bic to be a new instance of class Pen. Pen is a class of objects that behave like plotter pens. Pens understand how to change direction, lift up or press down on the drawing surface, and move.

new is the instance creation message sent to the class.

bic mask: Form black.
This line specifies that a black form be the mask for the pen bic.

black is a message sent to the class Form. An instance of class Form is an array of bits representing a rectangular region on the display. The Smalltalk system supports a bitmap in which pixels are either black or white. Form black creates a black instance of a Form.

The mask: message specifies that a black image be the mask for bic. A mask can be used to create halftone shades (such as the gray background of the display) by combining patterns of black and white pixels. You can think of a mask as corresponding to the shade of ink to use for the pen. In this case, the ink is black.

bic defaultNib: 4.
This line gives bic a nib four pixels wide.

4 is an object that the message defaultNib: takes as an argument. Any instance of class Pen understands the message defaultNib:.

bic combinationRule: Form under.
This line specifies what drawing mode the pen will use to draw on the display.

There are sixteen possible rules for combining source and destination pixels in order to draw on a surface. Each of these rules is assigned an integer from zero to 15. Several of the more useful ones have also been given message selectors (in this case, under) that are understood by class Form.

combinationRule: is a message that takes, as an argument, an integer specifying a drawing mode. The expression following combinationRule: must therefore evaluate to an integer. Form under returns the integer 7, specifying the combination rule for the inclusive OR. If either the source or the destination pixels were on, the destination pixel will be turned on. So the pen's black mask is OR'ed with the existing image.

`1 to: 50 do: [:i | bic go: i*4.  bic turn: 89]`
This line loops through the block fifty times. The block draws with the pen, and then changes its direction.

The block has one block variable, i. This is the index of the iteration loop.

There are two expression within this block.

```
bic go: i*4

bic turn: 89
```

bic go: i*4 moves bic in its current direction i*4 pixels. Because it is down when it moves, it draws. And because the index constantly increases, bic draws an ever-longer line as it loops. Each go: invocation draws one edge of the spiral.

bic turn: 89 alters the pen's direction by 89 degrees. This gives the spiral its squarish shape.

PROGRAMMING TOOLS

When you invoke the standard Smalltalk image, a number of windows appear on your screen. Other windows can be opened as needed. Each window is a useful programming tool. This subsection describes some of the tools available to the Smalltalk user.

System Browser

The System Browser is one of the most commonly used tools in the Smalltalk system. With it, you can access Smalltalk source code.

The System Browser is used to access hierarchically organized information about the classes in the system and the messages they define. It has five major panes: four small ones at the top and one large one below. From left to right, the top panes are:

- The class category pane.
- The class pane.
- The message protocol pane.
- The message selector pane.

The large pane below is called the text pane.

The class category and message protocol panes (the first and third panes) are aids to organization. They do not have any counterparts in Smalltalk code. They are there to help people remember where to find a particular class or message.

The class and message selector panes access Smalltalk code.

The text pane holds the source code for the classes, instances of classes, and the methods they execute. It also holds code templates that can be edited to add new classes and methods. Smalltalk does not visibly distinguish between code which was in the standard image, and code which you have added.

From each pane, sub-browsers can be spawned as needed to access information at any level of detail.

Workspaces

Workspaces are the scratch pads of the Smalltalk system. Workspaces allow you to try out code as you are developing it. In workspaces, temporary variables need not be declared.

Each workspace is independent of all other windows in the system. Any objects assigned to temporary variables in a workspace are local to that workspace. They are lost when the workspace is closed.

System Transcript

The System Transcript is a window used to log system messages. It is comparable to the *stdout* device on UNIX computer systems. It records your actions when you make a snapshot, for example, or when you file a piece of code in from a file residing on the disk. When you perform such an action, it is frequently useful to watch your System Transcript to verify that things are proceeding properly. If they are not, you may get informative error messages. When developing code, you should always keep a System Transcript window open and unoccluded by other windows.

System Workspace

The System Workspace contains many messages of general utility to Smalltalk users. These messages can be selected and executed as in an ordinary workspace. Unlike an ordinary workspace, however, the System Workspace can be closed and reopened; its contents are saved.

In the System Workspace, for example, there is code to change the text style of all your windows, code to customize your image and changes files, or templates to help you find all objects that implement a specified message. Even if you are familiar with previous versions of Smalltalk, spend a few minutes to familiarize yourself with the tools available from the System Workspace. They change somewhat from one release to the next.

File List

The File List is an interface to the file system. It allows you to access your directories and files as you would from the operating system. It also allows you to file in source code to modify your image, or to edit files and put the modified contents back.

Inspectors

Any Smalltalk object understands the message `inspect`. This message opens an inspector on the object. After framing the window, you can examine the contents of all instance variables and indexable fields of the object. For example, opening an inspector on an instance of a Collection allows you to examine the size, order, and contents of that collection.

Debuggers

The debugger is a Smalltalk tool helpful for debugging code. It shows you where execution of a process was interrupted.

The debugger is divided into two panes and two inspectors. The top pane is the list of message sends, with the most recent one at the top. When you select one, its method is shown in the second pane. The part of the method that was executing when the process was interrupted is highlighted. From there, you can step through the method one expression at a time.

The left inspector allows you to inspect the object receiving the message shown above, and the value of each of its instance variables. The right inspector allows you to inspect the current value of the arguments and any temporary variables for the method shown in the pane above.

For more information on the debugger, see Goldberg, Chapter 19.

EXTENDING THE LANGUAGE

Most work in Smalltalk involves adding new classes and methods. Using the concepts and programming tools presented so far in this section, you are now ready to add first a new method to an existing class, and then a new class.

The information below is presented in tutorial format. Turn on and log into your machine, invoke your image (or the standard image, if you prefer), and execute the steps as they are presented. This tutorial presumes that you have completed the previous tutorial in Section 2, or that you are comfortable with the user interface from your own explorations.

Both tutorials in this section consistently distinguish between *typing* and *entering* text. To *type* text is to use the keyboard to cause the characters to appear on the screen. To *enter* text is to perform the additional step of pressing <RETURN>, or using the accept menu item from the middle button menu, to enter your text into the system.

Adding a New Method

In this tutorial, you will add a new method to class String. This method will allow it to reverse the case of any alphabetic characters in a string. You will first open a workspace to write and test your code. After the code is debugged to your satisfaction, you will then install it in the Smalltalk system using the System Browser.

Workspaces are a good place to test new code, for several reasons.

- You can execute expressions one at a time, testing your new code incrementally.
- Temporary variables need not be explicitly declared, but they persist for the lifetime of the workspace and can be inspected after the code has been executed.
- You do not need to decide where, in the Smalltalk hierarchy, to place your new method until you have finished its design.

When you add code to the System Browser, you will discover a feature implemented to protect you from losing work. When you are adding text to the text pane, until the text is accepted you are unable to move around in the System Browser. If you try, the text pane flashes at you, and a notifier window pops up to warn you of loss of work if you change the view within the pane. Therefore, if you need to see other system code as you work, use the middle button menu command spawn to spawn another browser.

If you wish to see other system code as you work, some useful template expressions are available in the System Workspace. The expression Smalltalk browseAllImplementorsOf: #messageSelector under the heading **Inquiry** is particularly useful. Select the word messageSelector and replace it with any message selector you wish to inquire about. Leave the # sign, however. Then select the entire expression and execute do it. A new browser pops up, with two panes. The top pane contains a list of all objects that implement the message selector in question. If you select one of the lines in the top pane, the bottom pane fills with the method implemented by that object when it receives that message.

If you wish to take a break during this tutorial, or clean up your display, you can use the right button mouse menu to retitle your workspace and collapse it. Choose title and enter a new title in the window that pops up. Then choose collapse, and move the title tab out of the way for the moment. When you wish to resume your work, select the title tab and execute frame from the right button menu to get your workspace back again.

1. Open a workspace.
2. Type the following code in the workspace:

```
oldString ← 'JFMamJJasonD'.
newString ← String new: oldString size.
1 to: oldString size do:
        [:index |
          aCharacter ← oldString at: index.
          aCharacter isLowercase
              ifTrue: [newString at: index put: aCharacter asUppercase]
              ifFalse: [newString at: index put: aCharacter asLowercase]].
```

This is the algorithm that performs the case reversal on the characters of the string. A line-by-line explanation follows.

oldString ← 'JFMamJJasonD'.
This expression creates a string to experiment with. Type in any string you like. Be sure to enclose it within single quotation marks. It is assigned by the leftward arrow to the temporary variable oldString.

newString ← String new: oldString size.
This expression creates a new instance of class String. It is the same size as the original string. It is assigned by the leftward arrow to the temporary variable newString.

1 to: oldString size do:
This is the beginning of an iteration loop. The loop index starts at 1 and continues until it reaches the size of the old string.

The receiver of the message to:do: is an integer. Integers understand this message because they are subclasses of Number, from which they inherit the method. If you wish to examine the code for the message to:do: in Number, go to the System Browser, select Numeric-Numbers in the class category pane, and Number in the class pane. In the bottom of the class pane, you see the words class and instance. If you select class, the two rightmost panes allow you to access messages sent to the class only. If you select instance, the two rightmost panes allow you to select messages sent to instances of the class. Select instance, and select intervals in the message protocol pane. Select to:do: in the message selector pane and look at the code that appears in the text pane. This code evaluates the following block for the interval specified.

```
[:index |
```

The square bracket indicates the start of a block. This block needs a block variable, an index, to go through the loop. The index is not needed after the block is executed, however, so the variable need not be declared at the beginning of the method.

```
aCharacter: oldString at: index.
```

At this point, we need another temporary variable, aCharacter, to hold each character of the string as we execute the loop.

```
aCharacter isLowercase
        ifTrue: [ ]
        ifFalse: [ ]
```

isLowercase is a message that all instances of class Character understand. The statement evaluates to a Boolean true or false. ifTrue:ifFalse: is a message which these Boolean objects understand. It requires blocks as arguments. The appropriate block is executed, depending on whether the character at that index is upper- or lowercase.

```
ifTrue: [newString at: index put: aCharacter asUppercase]
ifFalse: [newString at: index put: aCharacter asLowercase]].
```

These lines perform the case reversal. If the character is lowercase, the corresponding element of the new string is made uppercase, and vice-versa. All instances of class Character understand the messages asUppercase and asLowercase. All instances of the class String understand the message at: put:. If you want to see the code for this message, look for it in the System Browser under the category Collections-Text, and the class String.

The second line also ends, with the second closing square bracket, the block that started after 1 to: oldString size do:.

In the System Browser, examine the methods for the messages being sent in this example until you feel comfortable with them.

3. Now test the new code. Select the entire set of expressions and execute do it.

4. If you received any error messages as you performed the last step, the system will draw your attention to the part of the code where execution was interrupted. Choose the abort command, and check to see that you have typed in the colons and periods as they appear in the code above. Then try again.

5. Type newString, select it, and execute print it to see the value stored in newString. The results of the print it command appear, selected, in your workspace. (New text appears selected to make it easy for you to cut, to clean up your workspace.)

   The new string reverses the case of the alphabetic characters in your old string. Nonalphabetic characters remain unchanged. If you look in the System Browser at the methods that implement the messages asUppercase and asLowercase, you see why.

6. Try your new code on as many strings as you wish. When you are satisfied that the code works as it should, you are ready to add the new method to the system. This is done in the System Browser.

7. Keeping your workspace open, activate the System Browser.
8. In the Class Category pane, the leftmost pane of the System Browser, choose the category Collections-Text.
9. In the class pane, choose the class String.
10. Select instance in the bottom of the class pane.
11. In the message protocol pane, bring up the middle button menu. It gives you the opportunity to add a new protocol. Execute add protocol.
12. A fill-in-the-blank window pops up. Enter the new protocol name reversing case. The text pane below now fills with a code template for you to edit.
13. In the bottom text pane, select message selector and argument names and replace it with the name of your new message selector — in this case, type asReverseCase. For this method, which has a unary selector, no argument names are required.
14. Select the string comment stating purpose of message. If you double-click the left mouse button between the opening quotation mark and the first character (or between the last character and the closing quotation mark), you can replace the entire sentence, but leave the quotes. Type: Answer a string whose characters reverse the case of the receiver.
15. Select the words temporary variable names between the vertical bars. Smalltalk code encloses temporary variable names in this way, with one or more spaces between each variable. In the algorithm you just tested in the workspace, there were two temporary variables: newString and aCharacter. (They did not need to be specifically declared in the workspace, but they must be now that the code is being added to the system.) Type the two temporary variable names.
16. Now select the word statements and cut it. Here is where you can copy the algorithm from your workspace, but you will have to make a few changes.
17. Go back to your workspace, and select all the text in it. Choose copy from the middle button menu.
18. Go back to the text pane in the System Browser. Choose paste from the middle button menu. Paste the text after the temporary variable declarations.
19. Delete the line that assigns a string to the variable oldString. The method you are now completing is used when the message asReverseCase is sent to any string. The receiver of the message takes the place of the variable oldString.
20. In Smalltalk, the receiver of a message can be referred to within a method by using the pseudovariable self. Replace the remaining three incidences of oldString with self.
21. Put the upward arrow ↑ before the last line, the line that says newString. You are specifying that your method return the value of the variable newString. An expression that returns a value must always be the last expression in the method or block, and must not be followed with a period. The last line now reads: ↑ newString

22. From the text pane in the System Browser, execute the middle button menu command accept. When you do this, several things happen.
    - The compiler evaluates the new code.
    - Your new message selector asReverseCase appears in the message pane.
    - Your method is now a part of the Smalltalk system. No distinction is made between code you have added and code that was already in the system.
    - Any new instance of the System Browser you create from now on will include the new method.

    If you wish to modify the method, make the desired changes and execute accept again. You may repeat this process as many times as you require.
23. Go back to the workspace (or make a new workspace) and type a new string. Be sure to enclose it in single quotes.
24. Now send it the message asReverseCase. For example, type:
    'i LIKE sMALLTALK.' asReverseCase
25. Select the expression you have just typed and execute print it. The new string appears, selected, in your workspace. Try it with as many strings as you wish.
26. If you save your image now, the new method is permanently added to it. If you do not want to add this new method to your image, quit now, without saving your work.

Adding a New Class

Classes in Smalltalk are situated in a hierarchy. Class Object is the root superclass. All other classes are subclasses of class Object.

In the tutorial above, you were able to send messages like to:do: to an integer because it had inherited the ability to respond to that message from its superclass. Such inheritance is a general characteristic of Smalltalk. *Inheritance* means:

- Any variable defined for a class is accessible to all instances of its subclasses.
- Any method implemented in a class is available to all instances of its subclasses.
- Any message selector responded to by instances of a class is responded to by all instances of its subclasses.

When you make a new subclass, you add to the system:

- A new class name.
- Additional class and instance variables (optionally).
- Additional methods (optionally). Existing methods may be overridden, thereby effectively removing them from the subclass.

Each subclass progressively refines the purpose and capabilities of its superclass. *Refinement* means:

- Each subclass takes maximum advantage of code in common with its superclass.
- Each subclass adds only those methods which are its specific reason for existing.

The Smalltalk language uses inheritance and refinement as powerful programming concepts. When you add a new class, consider carefully where, in the Smalltalk hierarchy, it should be placed. A sensitive and intelligent use of the existing hierarchy gives you powerful leverage. Your class can inherit a surprising amount of useful code, thus minimizing the number of new methods you must write for it.

It is a good idea, therefore, to look through the System Browser at some of the classes in the standard image. Classes can be grouped in several conceptual categories:

- Arithmetic classes. Examples of such classes include Integer, Random, SmallInteger, Fraction, Float.
- Data Structure classes. Examples of such classes include Set, Bag, LinkedList, Dictionary, Array, OrderedCollection.
- Programming Environment classes. Examples of such classes include Compiler, CompiledMethod.
- Graphics classes. Examples of such classes include Point, Line, Rectangle, Pen, Form.
- User Interface classes. Examples of such classes include Cursor, View, PopUpMenu, Browser.
- Communications classes. Examples of such classes include FileStream, FileDirectory, UTekSystemCall.

The middle button menu in the Class pane of the System Browser includes the command spawn hierarchy. You can select any class and execute this command to get a full list of its super- and subclasses.

In this tutorial, you add a new class, Event. This class stores certain information about events. In this tutorial, for the sake of simplicity, Event is a subclass of class Object. At the end of the tutorial, when the process is more familiar to you, a suggestion is provided for another new class. This class will reside lower in the hierarchy.

This tutorial presumes that you have completed the previous tutorial in this section. In this tutorial, you will perform the following steps.

1. Add the new class to the system.
2. Add instance protocol to the class. Instance protocol allows you to get the values of instance variables, to change those values, and to perform any other operations upon them that you require.
3. Add class protocol to the system. Class protocol allows you to create new instances of the class, and to perform other functions with the class as you may require.
4. Test the new class. In a workspace, create new instances of the class and use the instance protocol you have written to make sure that the code functions as you intended.

Ordinarily, you will test the new code in a workspace before adding it to the system. However, for the sake of brevity, start your work in the System Browser.

Define the New Class

The next steps add the class Event to the system.

1. Activate the System Browser.
2. Execute the middle button menu command add category. A window pops up, allowing you to enter your new category name. Enter the category name Sequenceable-Events. Ordinarily, you will select the appropriate class category in the pane. If one does not exist, create it as specified above.
3. A class definition template appears in the bottom pane. Edit it. Replace NameOfSuperclass with Object.
4. Replace NameOfClass with Event. Leave the # sign because the class definition message expects a symbol.
5. Select instVarName1 instVarName2 inside the single quotes (you can doubleclick at the beginning or end of the string) and type the instance variable names startingDate title duration. Separate them with spaces.
6. Select ClassVarName1 ClassVarName2 inside the single quotes and cut them. This class has no class variables. Leave the two single quotes, with no space between them.
7. This class uses no pool dictionaries, and the category is already filled in for you. Accept the text with the middle button menu.

   When you are done, your text pane will look like this.

   ```
   Object subclass: #Event
       instanceVariableNames: 'startingDate title duration'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Sequenceable-Events'
   ```

   The new class is now created, and must be commented.
8. The name Event appears in the class pane. Select instance, below. The middle button menu in the class pane includes the command comment. Execute it. The text pane fills with a new template.
9. Replace This class has not yet been commented with Instances of this class represent an event lasting a certain number of days. Accept the text.

Add New Instance Protocol

You are now ready to add instance protocol: the message categories for instances of your new class.

1. Make sure that instance is still selected in the bottom of the class pane. Choose add protocol in the middle button menu of the message protocol pane.
2. Enter accessing. You will write methods to get and return the values of instance variables.

3. You now have more choices in the middle button pop-up menu in the message protocol pane. Choose add protocol again and enter comparing. You will write a method to compare events.
4. Repeat the procedure and enter private. You will write methods to change the values of instance variables. This protocol is called private because it should not be called by objects belonging to other classes.
5. The new instance protocols appear in the message protocol pane. The names you have entered for them are conventional Smalltalk instance protocol names. Select accessing. A method template appears in the text pane.
6. Edit the template. Replace the words message selector and argument names with startingDate.
7. Replace the generic comment with Return the starting date of the receiver. Make sure it is enclosed within double quotes.
8. Cut the line for the temporary variables, as this method needs none.
9. Replace the word statements with the line:
   ↑ startingDate

   The text pane now looks like this:

   ```
   startingDate
       "Return the starting date of the receiver."

       ↑ startingDate
   ```
10. Accept the text. The new message selector appears in the message selector pane, and the method has been added to the system. You can now use the text pane you have edited as your new template for adding the rest of the accessing protocol.
11. Add accessing methods for duration and title which return the duration and the title of the event, respectively. Use the text pane for startingDate and replace the message selector and object returned.
12. Add an accessing method for completionDate. Since completion date can be computed from the duration and the starting date, a separate instance variable is not required. Use the statement ↑ startingDate addDays: duration
    The method addDays: is inherited from class Date.
13. When you have added all four instance methods, go back to the message protocol pane and select private.
14. Edit the template. Replace the words message selector and argument names with startingDate: aDate. Notice the colon at the end of the message selector, indicating it is a keyword message.
15. Replace the generic comment with Change the starting date of the receiver. Make sure it is enclosed within double quotes.
16. Cut the line for the temporary variables.
17. Replace the word statements with the line:
    startingDate ← aDate

The text pane now looks like this:

```
startingDate: aDate
    "Change the starting date of the receiver."

    startingDate ← aDate
```

18. Accept the text. The new message selector appears in the fourth pane.
19. Repeat the same process for duration and title. duration gets numberOfDays for its argument, and title gets aString.
20. Now select comparing in the message protocol pane.
21. Edit the template. Replace the words message selector and argument names with overlaps: anEvent.
22. Replace the generic comment with Returns a boolean — true if the receiver's time span overlaps the time span of an event, false otherwise.
23. Replace temporary variable names with the variables earlier later. Make sure they are enclosed within vertical bars.
24. Replace the word statements with the lines:

```
self startingDate < anEvent startingDate
      ifTrue:
         [earlier ← self.
         later ← anEvent]
      ifFalse:
         [earlier ← anEvent.
         later ← self].
↑ earlier completionDate >= later startingDate
```

Accept the text. Now the method for implementing the message overlaps: has been added to the system.

Add New Class Protocol

So far, the protocol you've been adding is for instances of class Event. Each class requires class protocol as well. Class protocol includes such categories as instance creation, so that each class can create new instances of itself.

1. Select class in the bottom of the class pane, and add the protocol instance creation. Proceed as you did with instance protocol creation.
2. Edit the template in the text pane. Replace the words

   message selector and argument names

   with

   newDay: aDay month: monthSymbol year: aYear title: aString duration: anInteger
3. Replace the generic comment with Create a new event with title aString and a duration of anInteger number of days.

4. Replace temporary variable names with the variable anEvent.
5. Replace the word statements with the lines:

```
anEvent ← self new.
anEvent startingDate: (Date
            newDay: aDay
            month: monthSymbol
            year: aYear).
anEvent title: aString.
anEvent duration: numberOfDays.
↑ anEvent
```

Accept the text. Now the method for creating a new instance of class Event has been added to the system.

Test the New Class

The new class and methods have been added to the system. Now you can make sure that everything works as you wish.

1. Open a workspace.
2. Create a few events. For example, type:
   theAAAI ← Event newDay: 13 month: #July year: 1987 title: '1987 AAAI' duration:5.

   birthday ← Event newDay: 15 month: #July year: 1987 title: 'birthday' duration:1.
3. Select each new event and execute do it.
4. If you would like to see the new events you created, type and execute theAAAI inspect.
5. Check to see if one event overlaps another. Type:
   theAAAI overlaps: birthday
6. Select this line and execute print it.

You have now added a new class to the Smalltalk system, and confirmed its functionality. If you feel ready for a more challenging exercise, you can use this new code to create another new class. Call it EventList. Create it as a subclass of SortedCollection. This class is under the category Collections-Sequenceable in the System Browser. In this way, you will inherit the instance variables first and last, and indexable fields. Each indexable field can hold an event.

1. Write one class method to create a new instance of EventList.
2. Write instance methods to:
   - Add an event.
   - Delete an event.
   - Sort the events in the event list by starting date, completion date, title, and duration.
   - Find and print all overlapping events.
   - Find and print all events that overlap a specified event.

After you have learned about the Model-View-Controller paradigm, you may wish to use this code as the basis of a personal calendar.

You have now learned how to add your own code to the Smalltalk system. Congratulations!

Section 4

User Interface Features

INTRODUCTION

This section documents the user interface features that are standard in Tektronix Smalltalk. There are ways to interrupt processes, use the menus, manipulate files, edit text, and control the display.

Objectives

The user interface concepts are most easily learned by trying them out for yourself in Smalltalk. The interrupts and exits provide a good way to explore Smalltalk methods in action or debug executing code. The menus can be explored by entering the appropriate window, bringing up the menu, and trying out the appropriate selection.

GLOBAL KEY COMBINATIONS

These key combinations have special meaning to Smalltalk and are available globally whenever Smalltalk is running.

Interrupts and Exits

The following key combinations give you the ability to interrupt running Smalltalk code and provide ways to exit immediately.

The first key combination, <CTRL–C>, creates an interrupt and allows you to invoke the debugger.

The next key combination, <CTRL–SHIFT–C>, is useful when menus don´t work and you can´t execute code in a window. <CTRL–SHIFT–C> invokes the Emergency Evaluator, allowing you to type expressions to restore control or exit Smalltalk. Some useful expressions are listed below.

If the keyboard and mouse are unresponsive, it may be necessary to press <CTRL–SHIFT–BREAK>, followed by a <CTRL–C>. This forces an immediate exit from Smalltalk (without creating any new Smalltalk processes) and returns control to the UTek environment.

CTRL-C

This is useful for aborting an action (e.g. getting rid of a prompter) or debugging code that is running. When you press <CTRL> and <C> at the same time, it interrupts the current process and causes a User Interrupt notifier to be displayed. This notifier displays the execution stack as a sequence of message sends. To terminate the process, choose the right-button command close.

To continue the process, choose the middle-button command proceed. To invoke the debugger, choose the middle-button command debug. This brings up a debugger view with facilities for examining the execution stack and correcting the methods and data. (For more information on the Debugger, refer to Section 5 of this manual and Section 19 of the Goldberg book.)

CTRL-SHIFT-C

Pressing <CTRL>, <SHIFT>, and <C> at the same time invokes the Emergency Evaluator, which appears as a black rectangle at the top of the screen with the words "Emergency Evaluator (priority 5) - - type an expression terminated by ESC". All of the code you type until you press <ESC> will be executed at Priority 5, allowing your code to override user interface processes. Although simple editing using the <BACKSPACE> key is allowed, the Paragraph Editor is not active.

Here are some useful expressions. Choose the one appropriate for your situation:

<ESC>
: Pressing this key immediately returns you to the calling Smalltalk process.

Delay initialize <ESC>
: Use this when the timing management is broken.

ScheduledControllers searchForActiveController <ESC>
: This looks for the window that wants to be active. This is useful for restarting the search for an active window.

Smalltalk snapshotAs: 'imageName' thenQuit: true <ESC>
: This saves the current image as the specified imageName then quits Smalltalk.

Smalltalk quit <ESC>
: Control returns to the UTek operating system. The image is not saved.

CTRL-SHIFT-BREAK

This is a last resort that exits Smalltalk without saving. If you have tried <CTRL–C> and <CTRL–SHIFT–C>, and neither works, (due to buggy user code or a system error) it is probably time to leave Smalltalk.

Press <CTRL>, <SHIFT>, and <BREAK> at the same time. This control key combination is trapped by the UTek kernel and calls the UTek terminal emulator. You should see the terminal emulator cursor (if it does not appear, try panning all the way to the left and top of the display to make sure the cursor comes into view). Smalltalk is still active, so press <CTRL–C> to terminate Smalltalk and return control to the UTek operating system.

You can clear the screen with following UTek commands:

clear	This clears the display.
conset default	This clears the Smalltalk cursor.

<CTRL–SHIFT–BREAK> preserves the UTek file system, and should be used in preference to rebooting your machine. Exiting Smalltalk in this way may leave recoverable code in your changes file. Refer to **Restoring Lost Image Information**, in Section 6.

CAUTION

If the system doesn't respond to <CTRL–SHIFT–BREAK> followed by <CTRL–C>, it may be necessary to press the <POWER> or <RESET> button. This is a drastic measure, since resetting power could destroy your files.

Cursor Center Key

The cursor center key <F12> moves the cursor to the center of the visible viewport. Since systems with pannable (640 x 480) screens have virtual screens that are much larger than the visible screen, it is possible for the cursor to "disappear" off-screen. Pressing <F12> recenters the cursor.

If the cursor is unlinked when the key is pressed, it is re-linked. A portion of the cursor form is normally constrained to be within the bounds of the virtual screen. But, depending upon the shape of the cursor within its 16 by 16 pixel cursor Form, you may not always be able to see it.

MENUS

In Smalltalk, menus are the interface for finding information, and they are usually created in response to pressing a mouse button.

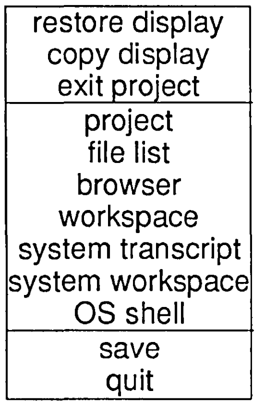
The left button on the mouse is frequently referred to as the "red" button, and it selects information.

The middle ("yellow") button creates menus of messages which can be sent to the currently active view. These messages edit the contents of the view.

The right ("blue") button creates menus of messages that modify the view itself, not the contents.

System Menu

This menu is available when the cursor is over the gray background (not in a window) and the middle button is pressed.



When you press and hold down the middle button the following choices are shown:

restore display	This redraws the display, removing everything that is not known to the control manager. The cursor is reset to the default slanted arrow cursor.
copy display	This copies the screen bitmap to a file. You will be prompted for the file name.
	This file can be sent to a printer, and is in the same format as an object of the class Form.
exit project	Projects are collections of views of information; several can co-exist at once. Each active project pre-empts the entire display screen.
	To create a project, select project on this menu. If you then select enter with the middle button, you will be inside the project. These two commands can be repeated to nest projects within one another. When you select exit project, you move up one level. When you are at the level of the topmost project, exit project has the same effect as restore display.
project	This creates a new project view. You will be asked to frame the project view before it appears.
file list	This creates a new FileList browser. You will be asked to frame the FileList browser before it appears.
browser	This creates a new System Browser. The System Browser allows you access hierarchy-organized information about the Smalltalk system. You will be asked to frame the System Browser before it appears.
workspace	This creates a blank area named Workspace, where text can be edited. You will be asked to frame the Workspace before it appears.
system transcript	This creates a view of the System Transcript. You will be asked to frame the view before it appears.
system workspace	This creates a view of the System Workspace. This contains message evaluations that you can edit and evaluate, dealing with file access, system queries, crash recovery, and a number of other subjects. You will be asked to frame the view before it appears.
OS Shell	This pauses Smalltalk and allows the user to communicate with a shell by using a terminal emulator. If the calling shell is C-Shell, Smalltalk is

suspended and control is returned to the calling program. Otherwise, a new shell is created.

After selecting OS shell, if you are not sure which shell you are using, type *ps -x*. This will list the processes you are running. Your shell will listed as *csh* (C-Shell) or *sh* (Bourne shell).

If you´ve been using the C-shell, typing *jobs* will show Smalltalk as a stopped job under the name that originally invoked it. If it is the only stopped job, typing *fg* will bring it back into the foreground, restoring Smalltalk as it was. If there is more than one stopped job, typing *fg* by itself will simply restore the most recently stopped job, which may not be Smalltalk. Look at the *jobs* listing to find the bracketed job number *[n]* and type *fg %n*. This will restore Smalltalk.

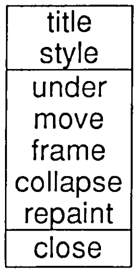
If you are not using the C-Shell, type *exit* or press <CTRL–D> to restore Smalltalk. This terminates the new shell and returns control to Smalltalk.

save
: This saves the current image of the Smalltalk system in a file. You will be prompted for the file name. Pressing <RETURN> uses the default file shown in the prompter.

quit
: This exits Smalltalk. You will be asked if you wish to save your image, or if you want to return to Smalltalk.

Standard System View - Blue Button Menu

This menu is available within all standard windows when the right button is pressed.



The choices on this menu manipulate the windows themselves, controlling the size, shape, position, and fonts within the active window. When you press and hold down the right button the following choices are shown:

title
: This allows you to retitle the currently active window; for example, you could retitle Workspace to FillInTheBlank.

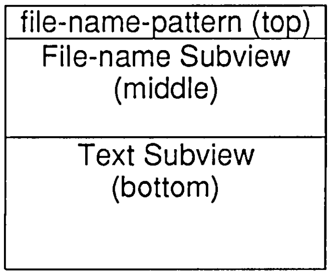
style
: This displays another menu which lets you select the font used within the current window (including subviews). It shows more text if you´re willing to read 8- and 10-point characters, or shows text with serifs if that´s what you like to read. The available text styles are determined by the contents of StyleManager. See the System Workspace for an example of how to add text styles to your image.

If the selected fonts have not been already loaded into your image, it may take a minute or so to load them from the disk.

under	This releases control from the current window and brings the window underneath into view. The new top window will become the active window.
	Windows overlapping each other are arranged in an internal stack. under pops the window on the bottom of the stack to the top. The cursor must be directly over the hidden window.
move	This allows you to change the position but not the size of the current window. When you initially move an uncollapsed window, the "top-left" cursor appears, which you can move around the screen. When it is in the desired position, depress the left mouse button and the window will be redrawn.
frame	This changes the size and position of the window by letting you switch between positioning the top-left and bottom-right corners.
	When you initially frame a window, the "top-left" cursor appears, which you can move around the screen. When it is in the desired position, press the mouse button and hold. The "bottom-right" cursor will appear. You now have two options:
	The first is to remove your finger from the left mouse button; this selects the rectangle just framed.
	The second option is for you to lift your finger from the mouse for an instant and immediately press it again. This moves the cursor back to the top-left corner of the rectangle, allowing you to re-adjust that corner. Lifting your finger and immediately pressing it again returns you to the bottom-right corner. You can toggle corners as long as you want. As soon as you lift your finger and leave it up the position and size of the window will be fixed.
	(The duration of the "instant" is made by an instance of class Delay, which is created in the StandardSystemView method getFrame. There is a constant in this method that specifies the waiting time at 250 milliseconds.)
collapse	This deletes from the screen all of a window except its title tab, which can then be moved. The contents of the window can be seen again by selecting frame.
repaint	This redisplays the contents of the current window.
close	This removes a window entirely from the display and deletes its structure.
	The contents of the window are discarded. close will discard the code in a workspace unless you file it away.

FileList Menu

An instance of FileList is created and opened by selecting the System Menu command file list. The FileList opens up with three subviews.



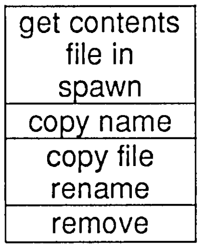
Typing text in the top subview specifies a pattern. This pattern may consist of completely or incompletely specified names that may or many not exist in the UTek file system.

The sequence of characters in the pattern may contain the wildcard characters <*> or <#>. <*> matches any number of characters and <#> matches any single character. These wildcard characters can only be used in the file name itself and not in the path name. For example, /usr/lib/smalltalk/fileIn/* and /usr/lib/smalltalk/fileIn/workspaceFileOut.st are acceptable, but /usr/lib/small*/junkfile is not acceptable.

When you choose the middle-button command accept in the top subview, the middle (file-name) subview displays all the names of the files or directories that matched the pattern in the top subview. When you select a name from the middle subview, a comment is displayed in the bottom (text) subview.

If you selected an existing file, the size and last modification date and time are displayed. If you selected a directory, a message states that the selected file is a directory. If a new file name was accepted in the top subview of the FileList, the comment – new file or directory – is displayed.

If the selected name is an existing file, the following menu appears when you press the middle mouse button:



get contents	This displays the contents of the selected file in the text subview of the FileList browser.
file in	This retrieves the entire contents of the selected file, reading and evaluating the text according to the file format for class definitions, methods, and expressions. A useful set of Smalltalk applications to "file in" are found in the UTek path /usr/lib/smalltalk/fileIn/*. You are invited to browse these and add the ones you like to your image.
spawn	This creates a new view that displays the contents of the file. This is similar to the get contents selection, with the difference that the file contents are displayed in a new File Model view. This simplifies cut-and-paste across multiple windows.

copy name	This copies the text of the file name into the text editor buffer. Although it is not visible, it is ready to be "pasted" into other text views.
copy file	This copies the selected file into a destination and name of your choice. A prompter will appear, allowing you to type in a new file name (and path, if necessary). Finish by pressing <RETURN> or choosing the middle-button command accept. If you type an unacceptable file name, or no file name at all, a confirmer will appear to determine whether you want to try again.
rename	This changes the name of the selected file. A prompter will appear with the file name of the selected file. Enter the new file name, press <RETURN>, and the menu of file names will be updated.
remove	This deletes the file name from the directory. A confirmer will appear to determine if you really want to remove the selected file. Choose yes to remove the file.

If the selected name is a directory, the choices offered are a little different, as shown below:

list contents spawn
copy name rename
remove

list contents	This lists the names of files within the directory in the bottom pane.
spawn	This opens a new FileList browser on all the files within the selected directory. The directory name appears in the top (pattern) subview.
copy name	This copies the text of the directory name into the text editor buffer. Although it is not visible, it is ready to be "pasted" into other text views.
rename	This changes the name of the selected directory. A prompter will appear with the directory name of the selected directory. Enter the new directory name, press <RETURN>, and the FileList browser will be updated.
remove	This removes the directory if it is empty. If it is not, a notifier appears saying that the directory cannot be removed because it is not empty. If this happens, select spawn and then remove the files within the directory.

If a new name is accepted in the top pane of a FileList and that name is selected in the middle pane, the middle button menu is:

copy name rename
new file new directory

copy name	This copies the text of the new file name into the text editor buffer. Although it is not visible, it is ready to be "pasted" into other text views.
rename	This changes the name of the selected file. A prompter will appear with the file name of the selected file. Enter the new file name, press <RETURN>, and the FileList Browser will be updated.

new file	This creates an empty file with the specified name and replaces the current middle button menu with the normal middle button menu for files (see above). A new file can also be created by putting text in the bottom pane of the FileList browser.
new directory	This creates a new directory with the specified name. The middle button menu is switched to the directory menu.

PARAGRAPH EDITOR

The Paragraph Editor uses commands from the middle-button mouse menu and commands from the keyboard. It is the primary tool to copy, cut, paste, execute, or accept text in the Smalltalk system. It uses the concept of a selected area of text which then has menu-selected operations performed upon it. The first part of this subsection describes the various means of selecting areas of text; the second part describes the various operations which can be done with this text.

Text Selection

Text is selected by using the moving the mouse and using the left button. When you move the mouse into a view of text and click the left button, a caret (^) appears at the cursor location or at the gap just before a character. To create a zero-width selection, click once between characters.

If you want to paste text, click the left button once at the intended insert point, then depress the middle button and select paste from the menu.

To select an area of text, move the cursor to your intended starting point. Depress the left button. Continue to hold the left button down while you move the cursor to end of the area you intend to select. (This activity is called *dragging* or *draw through*). When you reach the end, release the left button. The selected text is highlighted.

This text can then be copied, cut, evaluated, or stored by pressing the middle button and choosing a selection from the menu. It can also be cut by pressing the <RUB OUT> key.

Double-clicking (quickly clicking the left button while holding the mouse still) can select areas of different size, depending on the location of the cursor.

To select:

a single word	Double-click within the word. If the word is not at the edge of a delimiter, you can also double-click slightly before or after the word.
a line of text	Double-click at the beginning of the line (following the newline), or at the end of line (preceding the newline). This only applies if the lines are delimited by newlines.
all text in the view	Double-click at the beginning or end of the text in the view.
delimited text	Double-click slightly before the left delimiter or slightly after the right delimiter. (Delimiters will not be included in the selected area.) Acceptable pairs of delimiters are: [ ] ( ) < > " " ´ ´

To select all text that has been typed since the last mouse click, press <ESC>.

To select just slightly outside the boundary of the view, drag the cursor to the top or bottom of the view; automatic text scrolling will begin. Still holding the left button depressed, momentarily move the mouse in the opposite direction of the automatic scroll; this will stop the text from scrolling. Move the cursor to the intended selection boundary and release the left button; the selection will take effect.

Extended Selection

Extended selection is the ability to dynamically increase or decrease the amount of selected text; one of its uses is selecting regions of text that exceed the size of the window. Another use is to "fine-tune" a selected area that might have initially missed a few characters.

Select an initial area of text by dragging the mouse (moving the mouse with the left button depressed) over an area of text. Release the left button. The selected area appears in reverse video. To alter the boundaries of this area, move the cursor either within the selected area or outside of it, click <SHIFT–LeftButton>, and the selected area will either decrease or increase, depending on the location of the new selection. If you drag with the <SHIFT–LeftButton> depressed, you can resize an existing selection.

Once you depress just the <LeftButton> by itself, though, the selected area will become unselected and you will be starting a new selection.

Editing Key Combinations

Pressing <CTRL> and another key at the same time will perform different operations on selected text. These include cuts, inserting delimiters, and changes to the font. Two other keys, <RUB OUT> and <BACKSPACE>, perform a cut on the selected text.

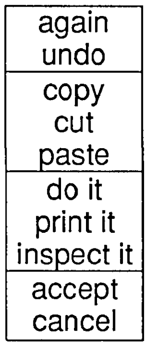
RUB OUT	This will cut the selected text, deleting it from the view and placing it in the text editor buffer.
BACKSPACE	Cuts the selected text and the character preceding the selection.
CTRL W	Cuts the selected text and the word preceding the caret (^). This is useful for erasing the previous word while typing.
CTRL D	Inserts the current date.
CTRL T	Inserts the text ifTrue.
CTRL F	Inserts the text ifFalse.
CTRL B	The selected text is emphasized as boldfaced.
CTRL SHIFT B	The selected text is changed to non-boldfaced.
CTRL I	The selected text is emphasized as italic.
CTRL SHIFT I	This selected text is changed to non-italic.

CTRL X	The selected text is emphasized as bold italic.
CTRL SHIFT X	The selected text is changed to non-bold non-italic.
CTRL E	Clears the emphasis (bold, italic, or both) from the selected text.
CTRL –	The selected is underlined.
CTRL SHIFT –	The selected text is changed to non-underlined.
CTRL 1	This changes the font of the selected text. The system default for **CTRL 0** is Pellucida San-Serif 8-point. The following **CTRL 2** through **CTRL 0** key combinations change the font of the selected text to the StrikeFonts listed below. To view the default StrikeFonts, open the System Workspace, look under Globals, find and select TextConstants, and make the middle-button selection inspect it. When the Dictionary view opens, select DefaultTextStyles and make the middle-button selection inspect. When the TextStyle view opens, select fontArray. The right-hand subview of TextStyle will display the array of StrikeFonts in the order shown here, starting at **CTRL 1**, going through **CTRL 9**, and ending with **CTRL 0**.
CTRL 2	Pellucida San-Serif 8-point bold.
CTRL 3	Pellucida San-Serif 8-point italic.
CTRL 4	Pellucida San-Serif 8-point bold italic.
CTRL 5	Pellucida San-Serif 10-point.
CTRL 6	Pellucida San-Serif 10-point bold.
CTRL 7	Pellucida San-Serif 10-point italic.
CTRL 8	Pellucida San-Serif 10-point bold italic.
CTRL 9	Pellucida San-Serif 8-point underline.
CTRL 0	Pellucida San-Serif 8-point bold underline.
CTRL [	If the selection is currently bounded by the [ ] delimiter pair, they are removed. If the selection is not bounded by this delimiter pair, they are inserted.
CTRL (	( ) delimiter pair, as above.
CTRL <	< > delimiter pair, as above.
CTRL "	" " delimiter pair, as above.
CTRL ´	´ ´ delimiter pair, as above.

Text Editing Menu

Pressing the middle button when the cursor is in a view of editable text creates the Text Editing menu. These commands operate on the selected text in the view.

Most of these menu items are available in views of editable text. Some of these are available in specialized views.



again	This does the last copy, cut, or paste command again.
undo	This will undo the effect of the last command (if it can be reversed).
copy	This copies the currently selected text in the text editor buffer.
cut	This deletes the currently selected text and places it in the text editor buffer.
do it	This evaluates the currently selected text as a Smalltalk expression. Syntax errors are checked during initial compilation.
print it	This evaluates the currently selected text as a Smalltalk expression. In addition, the result of the evaluation is appended to the text immediately after the original selected text. The evaluation result then becomes the current selection.
inspect it	This opens an inspector on the evaluated result of the selected code. This has the same effect as appending the word inspect to your code, and then choosing do it. (This command only applies to workspace views.)
accept	This stores the currently selected text. The meaning of this is context-sensitive. If the text was created in a workspace, it is stored; if the cancel command is used later, this stored text replaces the current text. If the text was created in a browser, it is compiled. The compiled method is stored if the compilation is successful.
cancel	This restores the text in the view to the condition of the last accept command. If no accept command has been given, the text is restored to the condition it was in when it first appeared in the view.

file out is an unsupported Smalltalk "goodie" which you can "file-in" to your image. It is in the UTek path /usr/lib/smalltalk/fileIn/workspaceFileOut.st. After it is "filed-in", it appears as a menu selection on the text editing menu when the cursor is in a workspace.

file out opens a prompter, which asks you for a new file name for your workspace contents. After you type in your file name, it copies the workspace contents into the new file, and appends code that will create a new workspace when it is "filed in" again. file out is a simple way to store a workspace in a file.

WINDOWS

Windows, or *views*, provide areas on the screen to view information. These windows can be moved, resized, collapsed to their title tabs, moved over each other, or deleted. If the view on the screen is too small to display an entire document, *scroll bars* can be used to "scroll" the view through the document.

Scrolling

When you move the cursor beyond the left part of List and Text views, the cursor enters the scroll area. The scroll area provides information about the view and provides the ability to "scroll" the view.

The gray marker approximately represents the viewable part of the contents. The size of the marker compared to the size of the scroll bar represents the percentage of the contents that are viewable; similarly, the position of the marker represents the location of the view.

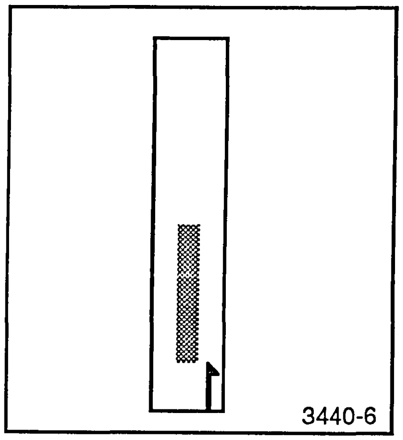


Figure 4-1. Scroll Next (relative move).

For example, enter the System Workspace and move the cursor into the scroll bar, where it becomes an arrow. Move it slightly until it becomes an up-pointing arrow. (See Fig. 4-1). If you move this up-pointing arrow to the very bottom of the scroll bar, then click any button, the text will move one page forward.

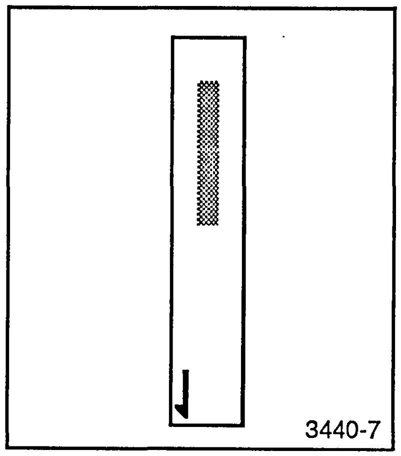


Figure 4-2. Scroll Previous (relative move).

Next, move the arrow left until it points downward, keeping it a the bottom of the scroll bar. (See Fig. 4-2). Click any button. The text will move one page backward (where you started). When the arrow points up or down, it will move *relative* to the current location. If the arrow is at the bottom of the scroll bar (and pointing vertically) it will make the largest possible relative move – one page. As it gets closer to the top, the relative moves get progressively smaller, ending with one-line moves when the arrow is at the top of the scroll bar.

Continuous scrolling is available when you are making relative moves; just hold the button down while the arrow is pointing up or down. The closer is the cursor is to the bottom of the scroll bar, the faster your text will scroll.

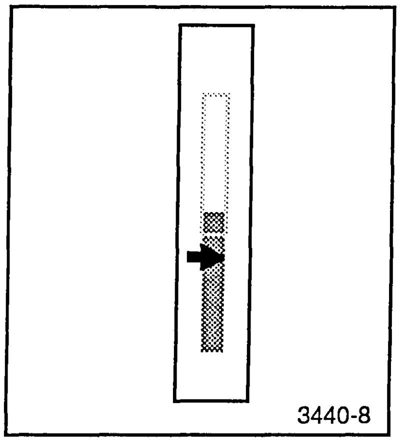


Figure 4-3. Jump (absolute move).

Next, move the cursor slightly so that a right-pointing arrow appears. The arrow now indicates the desired location of the text in the workspace, with the arrow itself representing the center of the text. This arrow controls *absolute* motion. If the right-pointing arrow is at the top of the scroll bar and you click any button, you'll see the top of the workspace. If you move the arrow to the middle of the scroll bar and click the left button again, you'll see the middle of the workspace.

If you put the right-pointing arrow into the gray marker and hold the button down, you can drag the text dynamically as you move the marker up or down.

Background

If you want to change the background behind your windows, you need to change the form which defines the background. Open a workspace and enter the following line:

```
ScreenController backgroundFromUser
```

Select this line and make the middle-button menu selection do it. This lets you select any area of the screen, making it a Form, which is then tiled repeatedly until it fills the entire screen background. Selecting do it on ScreenController whiteBackground, then ScreenController darkGrayBackground, then ScreenController GrayBackground lets you explore different screen backgrounds.

If you to explore further, open the System Browser and look in the ScreenController class (within the Interface-Support class category), which defines the backGroundFromUser method.

Repainting Windows

Windows do not unnecessarily repaint themselves. When a window is moved or closed, the parts of the visible windows immediately underneath it are repainted. The repaint menu item in the right button menu causes a window to be repainted.

By default, each time a window is de-emphasized, the visible contents of the window are saved in a form. This form allows the window to be displayed quickly the next time it becomes active. (The System Transcript window is an exception to this convention since it can be modified while inactive.) Since each window has a corresponding form, the existence of many windows can use a considerable amount of memory. An option is provided to turn off the default storage of these forms. To exercise this option, execute the expression Smalltalk saveSpace: true.

In conjunction with the saving of forms for windows is the redisplay of parts of inactive windows which have been obscured. For instance, if one window were covering a second window and the first window is closed, then the part of the second window which was previously covered by the first window would be redisplayed. The saved form for the first window is used to redisplay the obscured portion. (In the case of redisplay of the System Transcript window, the obscured portions are colored white).

The existence of many obscured windows needing to be redisplayed can be time consuming. An option is provided to turn off the default redisplay of obscured windows. To exercise this option, execute the expression Smalltalk frills: false.

Several combinations of these two options exist. One may wish to save the window forms in order to display windows quickly, but not want to redisplay obscured windows. In this case, set both saveSpace and frills to false.

Another reasonable combination is to redisplay obscured windows but not save forms. Since the forms associated with windows are not saved, obscured parts of windows are filled in with a white form. This combination may be used by people who want to limit memory usage and yet want to know explicitly where all the windows are. In this case both saveSpace and frills would be true.

To return the system to its default state, execute the two expressions frills: true and saveSpace: false. You can find the alternate forms of these expressions in the System Workspace under Globals.

Programmer's Notes for Repainting Windows

A number of methods implement this functionality. Here are some of the more important methods – in the StandardSystemView class (Interface-Support class category):

- deEmphasize (deEmphasizing message category)
- deleteDisplayForm (saved window form message category)
- displayForm
- saveDisplayForm
- setStatusDisplayForm

- validDisplayForm

and the ControlManager class (Interface-Framework class category):

- discardCachedDisplayForms (initialize-release message category)
- restoreWithin: (displaying message category)

and the View class (Interface-Framework class category):

- deEmphasize:andClip: (deEmphasizing message category)
- deEmphasizeView:andClip:

Some applications which implement new selection functionality tied to the top view (like the Paragraph Editor) may need to implement their version of these methods (although such applications are rarely implemented) – in the View class (Interface-Framework class category):

- deEmphasize:andClip: (deEmphasizing message category)
- deEmphasizeView
- deEmphasizeView:andClip:

Other applications which update passively scheduled windows (not the active one) will want to use the update: message in the View class (Interface-Framework class category).

- update: (updating message category)

The update: message throws away the saved bitmap of the windows. This is useful when the bitmap has become obsolete and needs to be replaced, but the window has not become active.

Section 5
Programming in Smalltalk

INTRODUCTION

This section covers several topics of interest to Smalltalk programmers. If you have programmed in Smalltalk on another machine, it explains some of the characteristics of the Tektronix Smalltalk programming environment. If you are just beginning to program in Smalltalk, it discusses several tools and philosophical issues that can have an effect on your approach to the language.

WORKSPACE VARIABLES

Workspace variables are variables that are automatically created whenever you assign a value to a name in a workspace. For example, if you type the following code into a workspace and do it, it creates a new object and assigns it to the variable named myObject.

```
myObject ← Object new
```

You can now manipulate the newly created workspace variable, myObject, from the workspace in which it is created. It will persist until you close that workspace, then disappear. Workspace variables are local to the workspace in which you create them — they are not accessible from outside the workspace.

Most other text panes in the Smalltalk system use temporary variables. For example, the text pane of an inspector does not have workspace variables and you must use temporary variables. Temporary variables persist only for the duration of the do it for which they are defined.

A workspace is the safest place to experiment in Smalltalk. With workspace variables, you can use a workspace to prototype code and execute and test it reasonably safely. Once your code works, you can move it into the system browser and modify it to integrate it into your image.

USING THE SYSTEM BROWSER

The System Browser is your primary tool for reading and developing code in the Smalltalk system. You use it to browse (read source code), alter source code, and add new code to your image. The browser is a sophisticated, easy to use, tool that can save many hours of effort.

Creating New System Browsers

You create a new instance of a system browser by placing the cursor on the screen background and selecting the browser item from the middle button menu. Some Smalltalk programmers often have several system browsers open at once, one to incorporate new code into the system and others to browse source that they may borrow or modify and re-use.

Anatomy of the System Browser

The System Browser is a five-paned window. The four panes across the top give you hierarchical access to Smalltalk source. The larger pane at the bottom is a text pane where the system displays source code and where you can write and modify source code. (Although the text pane is similar to a workspace, it's not a substitute for a workspace — it doesn't have workspace variables.) From left to right, the four smaller panes across the top of the System Browser are:

- The Category Pane
- The Class Pane
- The Protocol Pane
- The Message Selector (or Method) Pane

Each pane has associated with it a middle mouse button menu that lets you perform operations appropriate to that pane. In addition, there is a two position switch marked "class" and "instance" in the bottom of the Class pane. Let's examine each pane and what the menu for each lets you do.

The Category Pane

The Category Pane gives you a listing of all categories in the Smalltalk system. Categories are a user convenience — they are supported to help you organize classes and applications. See the later discussion on programming style for hints on how to use categories.

If you have no items selected in the Category Pane when you press the middle mouse button, you get an abbreviated menu. The selections act like those of the full menu. The full menu for the Category Pane is:

```
file out
print out
spawn
------------
add category
rename
remove
------------
update
edit
```

The menu selections for the Category Pane are:

file out	Writes a file suitable for filing into another Smalltalk image. The file name has the extension "*.st*" after the name of the category and contains the classes in that category. For example, if you have a category named MyCategory selected, you will create a file named *MyCategory.st*. When you file in *MyCategory.st* into another image and update the browser, you'll have added the entire category to the new image.
print out	Writes a formatted text file containing the Smalltalk source for the selected item. Printing out MyCategory writes a file named *MyCategory.pp*.

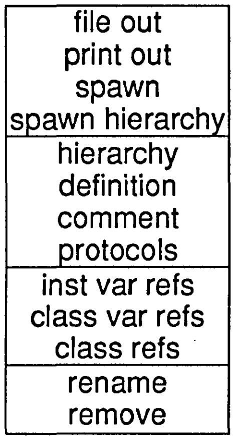
spawn	Spawns a category browser on the selected category. A category browser is similar to the system browser but has only classes in the selected category available. It does not have a category pane.
add category	Pops up a fill-in-the-blank. When you enter your new category name and accept or press the carriage return, the name is added to the Category pane. If you have a category selected, the new category is added after the selected category. If no category is selected, the new category is added at the end of the category list.
rename	Pops up a fill-in-the-blank with the selected category highlighted. Enter the new category name to rename the category.
remove	Removes the selected category and all its classes.
update	Tells the browser to redisplay its original model (Smalltalk). If you are using more than one browser and make changes to your image in one, other browser will not reflect the changes until you use update. If you file in code, it may not be reflected until you use update.
edit all	Puts the list of categories into the text pane for you to edit.

The Class Pane

The class pane shows a list of classes in the category selected in the Category Pane. If you have no classes selected, the pane will flash rather than give you a middle button menu.

In the bottom of the Class pane is a switchView pane with the labels class and instance. The active selection is highlighted. When the class selector is highlighted, the browser is examining the messages understood by the class as a whole. When the instance selection is highlighted, the browser is examining the methods understood by instances of the class. Although classes inherit methods, only methods defined within a selected class are displayed in browsers.

The menu for the Class Pane is:



file out	Files out a selected class. Filing out MyClass writes the file *MyClass.st.*
print out	Writes a formatted file containing the selected class. Printing out MyClass writes the file *MyClass.pp.*
spawn	Spawns a class browser (a browser containing only the selected class).
spawn hierarchy	Spawns a hierarchy browser containing the inheritance hierarchy of the selected class. A hierarchy browser contains no category or class list panes.
hierarchy	Displays the inheritance hierarchy of the selected class in the Text Pane.
definition	Displays the definition of the selected class in the Text Pane. The definition can be edited.
comment	Displays the class comment for the selected class in the Text Pane. The comment can be edited. Comments are usually defined on the instance side of the class.
protocols	Displays an editable list of the protocols and the methods they contain in the Text Pane.
inst var refs	Pops up a menu of the names of instance variables. Select one and you will spawn a browser on all methods that use that instance variable.
class var refs	Pops up a menu of the names of all class variables. Select one to get a browser on all methods that use that class variable.
class refs	Spawns a browser on all users of the selected class.
rename	Pops up a fill-in-the-blank with the selected class highlighted. Enter the new class name to rename the class.
remove	Removes the selected class.

The Protocol Pane

Protocols, like categories, are a user convenience. Smalltalk does not enforce a particular use of protocols. (See the later discussion on programming style for proper use).

If you have no protocol selected, the only menu item is add protocol. Otherwise, the menu for the Protocol Pane is:

```
 file out
 print out
   spawn
------------
add protocol
   rename
   remove
```

file out	Files out the protocol for that class. If you file out protocol1 in MyClass, the file will be called *MyClass-protocol1.st.*
print out	Writes a formatted file of the protocol (*MyClass-protocol1.pp*).
spawn	Spawns a protocol browser on the selected protocol.
add protocol	Pops up a fill-in-the-blank like the add category selection in the Category Pane. The new protocol will be added after the selected protocol, or at the end of the protocol list if no protocol is selected.
rename	Pops up a fill-in-the-blank with the selected protocol highlighted. Enter the new protocol name to rename the protocol.
remove	Removes the selected protocol and the methods it contains.

The Method Pane

The Method Pane lists the message selectors for methods under each protocol. You must have a selection to get a menu. The Method Pane menu is:

```
  file out
  print out
    spawn
------------
   senders
implementors
  messages
------------
    move
   remove
```

file out	Filing out method1 in MyClass writes the file *MyClass-method1.st.*
print out	Printing out method1 in MyClass writes the file *MyClass-method1.pp.*
spawn	Spawns a method browser on the selected method.

senders	Spawns a browser on all classes that send the selected method.
implementors	Spawns a browser on all classes that implement the selected method.
messages	Pops up a menu of all messages sent in the selected method. Selecting one of these spawns a browser on all classes that implement the message you selected.
move	Pops up a fill-in-the-blank that asks where you want to move the method. The current protocol is highlighted. This selection does not delete the selected method.
remove	Removes the selected method from the system.

The Text Pane

In addition to displaying the code for existing methods, the text pane shows the templates that you can edit when adding classes and methods to your image. The text pane, like a workspace, contains text editing menu items. Using do it or print it, you can send messages from the Text Pane of the browser, but the Text Pane does not have workspace variables; an attempt to send a message to an undeclared variable causes a notifier to pop up. In addition to the text editing menu items, items of particular interest are:

accept	This selection tells Smalltalk to compile the code in your Text Pane and place the result into your image.
cancel	This selection cancels any changes you have made to the text pane since the last accept.
format	Formats code for methods. To use, edit the contents of the Text Pane, accept your text, select format, then accept again.
spawn	If no method is selected in the Method Pane, spawns a new system browser. If a method is selected, spawns a method browser on that method.
explain	This selection tries to match selected text against tokens or constructs known to the system. If it finds a match, it returns a selected string that you can use as a key to more information. For example selecting explain on the word "explain" in the Text Pane returns the highlighted string:

"explain is a message selector which is defined in these classes (CodeController)." Smalltalk browseAllImplementorsOf: #explain

Notice that the explanation is in quotes — Smalltalk's comment mechanism followed by a Smalltalk expression. Since the text is already highlighted, select do it, and you can browse all implementors of the questioned text.

If the system cannot match the token or construct, it returns the string

"Sorry, I can't explain that. Please select a single token, construct, or special character."

PROGRAMMING TIPS

System Workspace Tools

The System Workspace provides templates for many expressions you may find useful to evaluate. The following four subsections describe some of the most useful expressions available from the System Workspace.

- The expressions under the heading *Inquiry* inquire about messages, methods, and literals in the system.
- The expressions under the heading *Globals* provide a list of global variables in the Smalltalk system.
- The expressions under the heading *Display* allow you to manipulate certain aspects of the display.
- The expressions under the heading *Measurements* allow you to ascertain the size or number of certain objects.

In addition, the expressions under the heading *Fonts and Text Styles* provide useful templates for dealing with the appearance of characters in your windows. Consult Section 9 of this manual for further information on this subject.

Finally, the expressions under the heading *Change Management* provide useful templates for manipulating your image and changes files. Consult Section 6 of this manual for further information on this subject.

Inquiry

The following expressions allow you to inquire about the relationships among the objects in the system. Many of these expressions require symbols, denoted by the # sign, as arguments.

Smalltalk browseAllImplementorsOf: #messageSelector
Replace the word messageSelector with the message selector you wish to inquire about. (Leave the # sign.) When you execute the statement, a browser appears containing all objects that implement the specified message selector. This expression is comparable to executing the menu item implementors in the message selector pane of the System Browser.

Smalltalk browseAllCallsOn: #messageSelector
Replace the word messageSelector with the message selector you wish to inquire about. (Leave the # sign.) When you execute the statement, a browser appears containing all methods that send the specified message selector. This expression is comparable to executing the menu item senders in the message selector pane of the System Browser.

Collection browseAllCallsOn: #timesRepeat
The message selector timesRepeat is provided as an example. You may replace it with any other message selector you wish to inquire about. When you execute the statement, a browser appears containing all methods in the class Collection that send the specified message selector.

Smalltalk browseAllCallsOn: #at: and: #at:put:
This expression allows you to access information about two message selectors simultaneously.

Smalltalk browseAllCallsOn: (Smalltalk associationAt: #Transcript)
When you execute this statement, a browser appears containing all methods that reference the global variable Transcript.

Smalltalk browseAllCallsOn: (TextConstants associationAt: #Centered)
When you execute this statement, a browser appears containing all methods that reference the literal Centered found in the pool dictionary TextConstants.

Smalltalk browseAllCallsOn: (Object classPool associationAt: #DependentsFields)
When you execute this statement, a browser appears containing all methods that reference the literal DependentsFields, one of Object's class variables. This expression is comparable to executing the menu item class var refs in the class pane of the System Browser.

Smalltalk browseAllSelect: [:meth | meth numLiterals > 40]
This expression opens a browser on all methods using more than forty literals. Any set of expressions may be substituted within the block. The browser opens on all methods such that, when the block is evaluated with a method as its argument, the result is true.

FileStream instanceCount
This expression answers the number of instances of the class FileStream that exists in the system at the time of evaluation.

FormView allInstances inspect
This expression creates an inspector on the collection of all the instances of the class FormView that exist at the time of evaluation.

Globals

The System Workspace lists all global variables in the standard image. You can add new global variables to the system by executing the expression:

```
        Smalltalk at: #GlobalName put: anObject
```

where GlobalName is the name of the new object you have created. GlobalName becomes a key in the Smalltalk dictionary. (Keys in dictionaries are symbols.)

You can remove global variables form the system by executing the expression:

```
        Smalltalk removeKey: #GlobalName
```

where GlobalName is the name of the global variable you wish to remove.

A number of global variables are necessary to run the Smalltalk programming environment. Below, each of these global variables is explained in greater detail.

Disk
This is the Smalltalk home directory. Any files you reference without fully qualified pathnames are treated as relative to Disk. This directory should be different for each image; otherwise, you will have one changes file for two images, which can cause problems. The specific directory each image uses for Disk is set above in the System Workspace, under the heading *Create File System.* Typing a * in the top pane of a File List matches all the files in your Disk directory.

Refer to Sections 6 and 11 of this manual for more information about the file system.

Display
This is the virtual display (by default 1376 by 1024 pixels). You can make a rectangle flash on the display by sending Display the following message:

```
        Display flash: (100@100 corner: 300@300)
```

Other useful messages that can be sent to Display are available from the System Browser, under the category Graphics - Display Objects, class DisplayScreen, instance protocol display functions.

Further messages to send to Display are in the System Workspace under the heading *Display* and described in a later subsection.

FontManager
This is the instance of StrikeFontManager used by the system. It contains instances of StrikeFonts.

A trikeFont contains bitmaps of characters to display on the screen, and information for displaying them. A trikeFontManager is a dictionary containing StrikeFonts.

The FontManager allows you to share fonts among text styles, saving memory. Some fonts may not be loaded into your image yet. If you inspect the FontManager, you see instances of VirtualStrikeFonts indicating unloaded fonts. As soon as you reference them from a TextStyle, however, they are automatically loaded.

Refer to Section 7 of this manual for more information about this.

ImageName
This is the word you enter to invoke your image. It is the string you type in the fill-in-the-blank window when you save your image. The default ImageName is image.

OS
This stands for *operating system.* It is a global variable used to avoid references to specific operating systems, for the purpose of enhancing code portability. Messages sent to operating system commands are sent to OS, which is set to the specific operating system used by your computer.

Processor
This is the scheduler for Smalltalk processes, such as activating a window or determining keyboard events. The Processor schedules higher priority processes preemptively. If you create two or more processes of equal priority, the Processor does not switch between them automatically. One of the processes must specifically relinquish control by sending the message:

```
Processor yield
```

ScheduledControllers
This is the window manager. It contains a list of all controllers in use by the project. It determines the order in which windows are stacked. The right button menu items under and restore display are implemented by ScheduledControllers.

Sensor
This is the instance of class InputSensor which keeps track of where the mouse cursor is. Some useful expressions to send to Sensor are found below.

Sensor cursorPoint returns the coordinates of the mouse cursor.

Sensor waitButton returns the coordinates of the point where the mouse cursor was when the left mouse button was pressed.

Sensor yellowButtonPressed returns true or false depending on the state of the middle button.

SourceFiles
This keeps track of the source and changes files for your image. It is an array. The first element

points to the source file. The second element points to the changes file.

StyleManager
This stores all available TextStyles. A TextStyle contains an array of fonts used by that style. It usually consists of two sizes of fonts, each in roman, boldface, and italic. You can access the StyleManager using the right button menu item style.

For more information about this, see Section 7 of this manual.

SystemOrganization
This determines the relationship between the class category and the class panes in the System Browser. If you inspect SystemOrganization, you see a categoryArray, which lists the categories in the leftmost pane of the System Browser. The elementArray lists the classes, in class category order. Stops tells the categoryArray how many classes, incrementally, are in each category.

Transcript
This points to the System Transcript of the current project. For a useful expression template, see the heading *Example use of a transcript*, below the list of global variables in the System Workspace.

In addition to the global variables, Smalltalk has *pool dictionaries*, also called *variable pools*. These are dictionaries containing variables available to a set of classes. A pool dictionary allows methods of those classes quicker and easier reference to all objects in that dictionary. Some important pool dictionaries are described below.

- Smalltalk is a pool dictionary available to the entire system. It stores all classes, global variables, and pool dictionaries. The compiler looks objects up in the Smalltalk dictionary.
- Undeclared is a pool dictionary containing any undeclared object. For example, if you file in a new method containing a name the system does not understand, the system places it in the Undeclared dictionary.

Display

The following expressions allow you to manipulate the display.

DisplayScreen displayExtent: 1376@1024
Make the virtual display 1376 by 1024 pixels. This is the default size. If you are using a machine with a smaller screen, you must pan to see the full display.

DisplayScreen displayExtent: 800@800
Make the virtual display 800 by 800 pixels. Using this expression template, you can make the display any size you wish.

NOTE

If you make the display too small to pop up a menu, you will be unable to continue work. You will have to exit your image without saving your work, and may lose some time.

Display setMouseBounds: Display boundingBox
This expression does not allow the mouse to move outside the bounds of the display. This is the

default condition.

`Display setMouseBounds: (-50@-50 corner: 1500@1500)`
This expression allows the mouse to move outside the bounds of the display. This can be useful if you wish to be able to scroll beyond the bottom of a window located at the bottom of the display.

`Sensor cursorPoint: Display viewportCenter`
This expression locates the cursor in the center of the visible display. Evaluating this is the same as pressing f12.

`Display getViewportLocation`
This expression returns the coordinates of the top left corner of the viewport. The *viewport* is the portion of the display which is visible.

`Display setInverseVideo`
This expression changes all black pixels on the display to white pixels, and vice-versa. Text appears as white letters on a black background.

`Display setNormalVideo`
This expression changes the display to the system default. Text appears as black letters on a white background.

Measurements

The following expressions allow you to inquire about the size and number of objects in the system.

`Smalltalk core`
Select this expression and execute `print it`. It returns the number of objects in the image at the time of evaluation, and the number of words they occupy.

`Smalltalk garbageCollect`
This expression forces a garbage collection. The Smalltalk garbage collector is a generation-based scavenging mechanism. Normally, it searches the image when Smalltalk needs memory, deleting objects not referenced by other objects. It looks at newer objects more frequently than older objects.

`MethodContext instanceCount`
This expression is a way to determine the number of interrupted executions of a method that are known to the system.

`Time millisecondsToRun: [SystemOrganization printString]`
This expression determines the length of time (in milliseconds) it takes to evaluate the expression(s) in the block.

```
MessageTally spyOn: [Behavior compileAll] to: 'spy.results'.
   (FileStream oldFileNamed: 'spy.results') edit
```
The first expression analyzes the performance of the expression(s) within the block, and writes the results to a file named *spy.results*. The second expression opens a window to examine the file in which the performance analysis has been stored.

Debugging

To aid in debugging code, Smalltalk has a debugger available as a standard tool. You can invoke a debugger whenever execution halts with a notifier. If you wish to force a notifier when code is executing normally, press <CTRL-C>. In the notifier, the middle button menu contains two items: proceed and debug. proceed continues the execution. debug brings up the debugger.

The debugger consists of six panes. The top pane presents the stack of message-sends that occurred just before execution was interrupted. The middle pane is a method browser. The bottom left pair of panes constitute an inspector on the receiver of the selected method. The bottom right pair of panes constitute an inspector on the arguments and temporary variables of the selected method.

The top pane of the debugger allows you to select each of the message-sends, in order to examine code and variable values. You can choose any message-send on the stack and cause evaluation to proceed from your selected point. You can also single-step through message-sends, checking the state of the variables to determine the source of the error. You can change the values of variables. If you evaluate expressions within the method browser pane, evaluation is carried out in the context of the currently selected method.

Each message-send displays the class of the receiver, and the selector of the message sent to the receiver. If the class in which the message was implemented is a superclass of the receiver, the class in which the interpreter found the method for this message selector is displayed in parentheses.

A selection in the top pane causes the corresponding method to be displayed in the middle pane. The methods can be edited and recompiled, using the accept menu item, just as in a browser.

Information appears in the method browser pane and the bottom inspectors only if an item is selected in the top pane.

Debugger Menus

Pressing the middle button form the top pane of the debugger brings up either of two menus. If no item in the top pane is selected, the following menu is available.

```
full stack
proceed
```

full stack	When a debugger is first created, at most only the top nine message-sends appear. Executing this command displays the complete stack of message-sends in the interrupted activity.
proceed	Executing this command closes the debugger and continues evaluation just after the interrupted point. The continuation assumes that the message at the point of interruption had completed and determined a value. The value for proceed is nil or the value of the last expression evaluated in the middle pane.

If an item is selected in the top pane, the following menu is available.

```
full stack
proceed
restart
return
------------
senders
implementors
messages
------------
step
send
```

full stack	When a debugger is first created, at most only the top nine message-sends appear. Executing this command displays the complete stack of message-sends in the interrupted activity. This is the same as in the previous menu.
proceed	The debugger closes and evaluation continues just after the interrupted point. The continuation assumes that the message at the point of interruption had completed and determined a value. The value for proceed is nil or the value of the last expression evaluated in the middle pane. This is the same as in the previous menu.
restart	The debugger closes and evaluation continues from the beginnning of the currently selected method.
return	Executing this command brings up a fill-in-the-blank window. This window allows the user to enter a value. The window closes and execution proceeds as if the selected method had returned the value that was provided by the user.
senders	This opens a message-set browser on all methods that send the currently selected message.
implementors	This opens a message-set browser on all methods that implement the currently selected message.
messages	This creates a menu of all messages sent in the method associated with the currently selected message. Choosing one opens a message-set browser to access all methods that implement it.
step	This evaluates the next message to be sent in the currently selected method. If no method is currently selected, it assumes the selection is the last message-send. After evaluating the next message-send, execution halts.
send	This command refines the above command step. If the next message to be sent were evaluated, it would consist of a sequence of messages. The command send is a request to enter the next message at the top of the activation stack, and be ready to evaluate the next message-send in its method.

Useful Debugging Expressions

When you are modifying methods or developing new ones, it is sometimes useful to insert pauses into your code to aid in debugging. The following expressions can be inserted into your code for this purpose.

- self halt
  Any object understands the message halt. This message stops the process when it is encountered, and a notifier pops up.
- self halt: 'Label string for notifier'
  If you wish to insert several pauses in the same method, you can specify a unique label for each notifier. In this way, you can determine which halt was encountered.
- Transcript cr; show: 'counter = ', counter printString
  Sometimes you may wish to print debugging information to the System Transcript. The line above shows an example. It first sends a carriage return, so that the information appears on a new line. Then it prints the string counter = , and then it prints the value of the counter at the moment the expression is encountered.
- (Delay forSeconds: 3) wait
  Sometimes you may not wish to print any information in the System Transcript. For example, if you are debugging code that displays text in a window, sending text to the System Transcript causes an infinite loop, as the code is used to display the text in the System Transcript window. In that case, you can use the expression above to delay for a specified number of seconds to indicate the point of execution. In this example, three was the number specified.
- Cursor wait showWhile: [Sensor waitButton]
  The line above can be used under the same circumstances as the previous line. It specifies that execution of the code will pause until the mouse button is pressed.
- Sensor leftShiftDown ifTrue: [*some sequence of expressions*]
  The line above allows you to execute code (including printing debugging information) conditionally. The code executes when the left shift key is pressed.
- self debug
  This line is the equivalent of Sensor rightShiftDown ifTrue: [self halt], which specifies that execution stop if the right shift key is pressed. This is useful, because the left shift key is sometimes polled in other code. The variant debug: 'Label string for notifier' can be used to label notifiers.

Some Cautions

This subsection contains a few words of caution about processes that others have found to be troublesome. When possible, we have tried to provide solutions to these problems as well.

Redefining the Equals Operation

The equals operation, represented by by =, is a fundamental operation. If you redefine it, you must be careful. Any two objects that are equal must return the same value for hash. Therefore, you may need to redefine hash as well.

To illustrate this, let's go back to the new class Event you made in Section 3. Suppose that you wanted to add comparing protocol to determine when two events were equal. You decide that two events are equal if they share a startingDate and a duration, and that their title is irrelevant. So you go to the System Browser and add the following instance protocol to the class.

```
= anEvent
   "Two events are equal if their starting dates and durations are the
    same.  Title doesn't matter."

    startingDate = anEvent startingDate and: [duration = anEvent duration]
```

You now wish to test this code. So you open a workspace and create the following two events.

```
party ← Event newDay: 12 month: #July year: 1987 title: 'party' duration: 1.

jaunt ← Event newDay: 12 month: #July year: 1987 title: 'jaunt' duration: 1.
```

In the same workspace, you create a dictionary to store your events.

```
aDict ← Dictionary new.
aDict at: party put: 1.
aDict at: jaunt put: 3
```

When you inspect the dictionary, you see that it contains the two events as two separate objects. Dictionaries use hash to compare objects, and you have not redefined hash. Because the title strings of the two events are different, hash sees them as different. They are therefore incorrectly stored as two distinct objects in the dictionary, even though you have defined them as equal.

To enable the system to consistently implement your definition of equality for an event, you must go back to the System Browser and redefine hash, as below.

```
hash
   "An event's hash depends on its starting date and duration.
   Title doesn't matter."

     startingDate hash bitAnd: duration hash
```

Now if you repeat the experiment with the dictionary in your workspace, you will discover that the dictionary contains only one object, the latter of the two to be added. Because hash now implements the same concept of equality as = for events, the dictionary correctly perceives the two events to be identical.

Changing the Position of a Class in the Hierarchy

Changing the position of a class in the hierarchy must be done with caution.

If the class cannot be recompiled in its new position, you will get a notifier. When the operation is aborted, the hierarchy may not be left intact. Usually, links pointing from the class you tried to move to its superclass remain. But links pointing from the (former) superclass to the class you tried to move must be reestablished by hand, in the following manner.

1. Inspect the former superclass of the class you tried to move.
2. In the left pane of the inspector, select subclasses.
3. The right pane of the inspector fills with a list of the subclasses. Check to see if the class you were moving is there.
4. If it is not, in the right pane of the inspector, type:
   `subclasses add: TheClass`
   where TheClass stands for the name of the class you tried to move.
5. Select the text you typed and execute the menu item do it.
6. Close the inspector.
7. Reopen the inspector to ensure that your operation reestablished the links.

Renaming Instance Variables

Renaming instance variables is a process consisting of several steps and thus prone to error, if you are not systematic. Rename instance variables by following these steps.

1. Add the new variable name without removing the old one from the class definition.
2. Recompile the class definition with the new variable by using the accept menu item.
3. Change all old instance variable references to the new one in every method.
4. Remove the old instance variable from the class definition.
5. Recompile the class definition again without the old name by using the accept menu item.

Another process may also be used if the class is not part of the standard user interface.

1. File out the class.
2. Remove the class from the image.
3. Activate a File List.
4. Get the contents of the file.
5. Edit the file containing the class definition in the File List.
6. Put the new contents back in the file, overwriting the old contents.
7. Substitute the new instance variable name for the old one globally.
8. File the changed class back into the image.

If the new variable name conflicts with a temporary variable name, problems will arise. You will need to reassign the instance variable a new name and repeat the process.

Closing a Window

In the course of developing an application, you may inadvertently create a window that does not fully know how to display itself or use menus. It may therefore be impossible to close the window using the right button menu. If this situation develops, here is an alternative method of closing a window.

1. Inspect all scheduled controllers by executing the following command.

   ```
   ScheduledControllers scheduledControllers inspect
   ```

2. The inspector brings up an ordered collection of StandardSystemControllers. In the left pane of the inspector, the item inspect is available from the middle mouse button. Inspect each StandardSystemController.
3. In the left pane of the StandardSystemController inspector, inspect its model to determine which StandardSystemController is controlling the troublesome window.
4. If the troublesome window is a workspace, you will not see the contents of any text that you have not accepted yet.
5. When you have inspected all the models and located the faulty controller, close the model inspectors.
6. Return to the inspector on the faulty StandardSystemController, and type the following commands in the right pane.

   ```
   ScheduledControllers unschedule: self.
   self close.
   ```

7. Select the two commands and execute do it. The window will close.

PROGRAMMING STYLE

Good code is clear, concise, and efficient. Good Smalltalk code is also brief, making much use of existing code, and adding only what it must. As with any other programming language, good Smalltalk style increases the programmer's productivity, reduces the number of errors, and improves system maintenance. Unlike most other programming languages, however, the source code in the Smalltalk system is available to all users of the system. Smalltalk code is accessed through a browser, a tool different from those used in traditional programming environments. Conventions for constructing identifiers, indentation, and placement of blank lines need to be adapted, therefore, to the Smalltalk programming environment.

Many new Smalltalk programmers have few established guidelines to follow. Because their code will influence subsequent programmers, it is important to establish the elements of good Smalltalk style. It should be easy to write good code if the programmer has these guidelines to follow.

The high reusability of Smalltalk code is another reason to adopt good stylistic practices. Programming by refining already existing code is one of the main paradigms in Smalltalk programming.

The following two subsections present some syntactic and semantic guidelines for achieving good Smalltalk style.

Syntactic Guidelines

The following guidelines deal primarily with syntactic issues, but the distinction is not always clear. Syntactic errors in formatting and punctuation may change the apparent meaning of an expression.

Comments

Class comments contain one or more paragraphs describing the general purpose of the class and how it is used. These paragraphs do not usually contain carriage returns so that the lines can be automatically formatted to fit the current size of the window.

To aid you in commenting your code, a class comment template has been provided in the standard image. When you make a new class, use the middle button menu item comment in the class pane. The comment template appears in the text pane.

Class and instance variables should be documented in the class comment. Each should be listed with its typical class and purpose. For example,

```
marker       <Rectangle> used to highlight the selection
tempNames <Array of: String> cached names of temporary variables
```

The initial comment in a method should describe the general purpose of the method. Be sure the comment fits the method. It is easy to copy a method, change it slightly, and forget to change the comment. A comment that does not agree with the code is worse than no comment at all, for it leaves the reader wondering which one represents the intended behavior.

If a method is typically called from a do it, provide a calling expression such as *"Pen example"* in a comment. If the method is short, this comment may appear at the beginning or the end of the method. If the method is long, include the comment at the beginning so the user will not have to scroll down to avoid missing it entirely.

If an invocation comment is placed near another comment, put each in a separate set of quotation marks or on a line by itself. This allows the invocation to be selected by doubleclicking inside a delimiter or at the beginning of the line.

```
example
            "Continually print two lines of text wherever you point with
            the cursor. Terminate by pressing any mouse button."
            "DisplayText example."
```

Capitalization and Variable Names

Capitalize shared variables: class variables, globals (including class names), and pool variables. Do not capitalize private variables: instance, temporary, and class instance variables.

When an identifier or message selector is composed of more than one word, the Smalltalk convention is to concatenate and capitalize all words but the first: basicSize findKeyOrNil:. Do not begin selectors or keywords with capital letters, except for proper names. The keyword selector copy:From:In: is therefore stylistically incorrect; copy:from:in: is preferred.

Do not abbreviate extensively. Smalltalk code uses full words, even when they are long, because it is open to everyone. The code must therefore remain readable.

Formatting and Indentation

Although formatting is one of the last things that many Smalltalk programmers worry about, it is one of the first things that impede the progress of anyone who examines the code later. The visual structure of the method should reflect the flow of control. Indentation should be used to illustrate this flow so that the bounds of the controlled block are clearly visible. Alternate cases should be indented at the same level.

To aid in formatting, the middle button menu in the text pane of the System Browser includes the item format. After you have accepted your text, use the format item to format your code in accordance with Smalltalk convention. The formatter also removes any unnecessary parentheses you may have included in your code.

Punctuation and White Space

With traditional programming languages, programs may be hundreds of lines long. Programmers are encouraged to use lots of white space to group similar parts and improve readability.

The direct equivalent, a Smalltalk program, does not exist. Smalltalk methods are generally shorter than their procedural counterparts: over 70% of the methods in the standard image are less than ten lines in length, including comments and blank lines.

Methods are accessed through the bottom pane of a browser. Programmers frame their browsers to achieve the best compromise between screen layout and ability to view methods of average size. Often they have to resort to scrolling. Most combinations of font, browser layout, and screen size result in a maximum of 35 to 40 lines of code visible, but the average is between 15 and 25. Extra blank lines and other unnecessary white space means that other code is hidden. Brackets or other delimiters on lines by themselves waste vertical space and are unnecessary if proper indentation is used. Doubleclicking just inside a delimiter can be used to match it.

If blank lines are used to separate sections of code in a longer method, do not use more than one. If they happen to fall at the bottom of a window, a reader may assume that the method ends at the break. A need to use blank lines to separate sections may also indicate that the method is doing more than one thing and should be split.

Unless grouping two short expressions in a block, do not use more than one complete statement per line. Many Smalltalk programmers might object to even this exception. It is easier to locate items, especially variable assignments, when they are aligned along the left margin. So

```
compositionRectangle ← compositionRect copy.
text ← aText.
textStyle ← aTextStyle.
firstIndent ← textStyle firstIndent.
rule ← DefaultRule.
mask ← DefaultMask.
```

is preferable to

```
compositionRectangle ← compositionRect copy.
text ← aText. textStyle ← aTextStyle.
firstIndent ← textStyle firstIndent.
rule ← DefaultRule. mask ← DefaultMask.
```

A single space on either side of assignment operators and binary selectors can also increase readability.

```
total←342   compared to  total ← 345
max<=1.23  compared to  max <= 1.23
```

In addition to better readability, a space between a keyword selector and its argument allows either to be selected efficiently with doubleclicking.

Spelling and Grammar

Proper spelling is as important in Smalltalk as it is in any system used by more than one person. Unfortunately, Smalltalk cannot as yet access an English language dictionary. Yet misspelling part of a message selector can create errors for someone who later tries to access the selector with the correct spelling.

Because users are most likely to see error notification messages, be particularly careful to check for spelling errors in these strings. If contractions are used, be sure to use two apostrophes in the string, so a single apostrophe will appear in the message. For example, the expression

```
self error: 'Can''t access timer.'
```

will appear as

```
Can't access timer.
```

Semantic Guidelines

This section lists guidelines for following established traditions, planning for future subclasses, structuring methods, organizing the class hierarchy, and establishing alternatives to nested conditionals.

Follow Established Conventions

To write good Smalltalk code, first read a lot of it. Follow guidelines already established in the code you read in the image.

Use the standard protocol names in the standard order so that they can be organized and located efficiently by others. When creating new subclasses of Controller, for example, use standard protocol names found in Controller. Organize them in the standard order: the initialization protocols such as initialize-release and class initialization are at the beginning of the list, and private is at the end.

For example, controlActivity, isControlWanted, and isControlActive are included in the Controller protocol control defaults. Suppose a programmer creates a protocol named control with a redefinition of controlActivity. Later, isControlWanted is copied from Controller into the subclass for a minor revision. In the standard image, the move menu item presents the user with a template of the class and protocol to copy to, initially Controller>control defaults. If the standard protocol name is used, only the name of the class would need to be changed. Inventing a new name means that the protocol will also need to be changed, or methods will be separated that are commonly classified together.

Experience also leads to the use of the most concise and common constructions. Although each of the following pairs of constructs produce the same results, the second set in each pair is preferable.

```
(Interval from: 1 to: 10) do:

1 to: 10 do:
```

Not only is the to:do: message shorter and more direct, it is one of the more commonly used messages in the system. The from:to: message is not called in the standard image and would be unfamiliar to most Smalltalk programmers.

```
aForm ← Form extent: 200@200.
aForm black.
aForm displayOn: Display at: 85@400.

Display black: (85@400 extent: 200@200).
```

Telling the Display to paint a region black with one line of code is preferable because it is shorter, simpler, and does not require a temporary variable.

Hide Implementation Details

Delegate the responsibility for carrying out implementation details. If implementation details change, the affected parts will be localized. For example, you may wish to check to see if two dates are equal. Instead of asking one date to compare itself with the other, the date asks its day to compare itself with the other date's day, and its year to compare itself with the other date's year.

```
= aDate
    "Answer whether aDate is the same day as the receiver."

    self species = aDate species
        ifTrue: [↑day = aDate day & (year = aDate year)]
        ifFalse: [↑false]
```

Nested Conditionals

The use of nested conditionals to implement case statements in Smalltalk means that abstract data structures need to be reexamined. In particular, the use of isKindOf: to decide which message to send to an object suggests a problem. In object-oriented languages, these conditionals can often be replaced by an effective use of polymorphism, so that the same message is handled differently by different classes. *Polymorphism* is the ability of different classes of objects to respond to the same message in different ways. This enables code to treat objects uniformly, even when they arise from different classes. The same message invokes different methods from objects of the different classes.

For example, the classes Date, SmallInteger, and String all implement the message selector = to compare instances, but the message selector is implemented by very different methods in each case.

Here is an example of a case statement used to map between two sets of data.

```
string = 'Bold'
        ifTrue: [char ← 'B']
        ifFalse: [string = 'Demi Bold'
                ifTrue: [char ← 'D']
                ifFalse: [
                        (some series of expressions)]]
```

A Dictionary can be used to replace the nested conditionals.

```
"StrikeFont class initialize
Dictionary associates face names with single character that symbolizes face"

FaceNames isNil
        ifTrue:
                [FaceNames ← Dictionary new: 8.
                FaceNames at: 'Bold' put: 'B'.
                FaceNames at: 'Demi Bold' put: 'D'.
                FaceNames at: 'Italic' put: 'I'.
                FaceNames at: 'Block' put: 'K'.
                FaceNames at: 'Oblique' put: 'O'.
                FaceNames at: 'Regular' put: ''.
                FaceNames at: 'Symbol' put: 'S'.
                FaceNames at: 'Bold Italic' put: 'X'].
```

If you wish to see how this dictionary is used, browse the method StrikeFont initializeFrom:.

Other times, nested conditional expressions can be replaced by returns. Here is part of a method (InputState keyAt:put:) that checks to see which key has been pressed.

```
KeyCodeIndex < 8r200
        ifTrue:
                [self normalKeyAt: index put: value]
        ifFalse:
                [index = CtrlKey
                        ifTrue: [ctrlState ← value bitShift: 1]
                        ifFalse: [index = LshiftKey
                                ifTrue: [lshiftState ← value]
                                ifFalse: [index = RshiftKey
                                        ifTrue: [rshiftState ← value]
                                        ifFalse: [index = LockKey
                                                ifTrue: [(some series of statements)]]]]
                metaState ← (ctrlState bitOr: (lshiftState bitOr: rshiftState))]
```

The nested conditionals can be replaced by calling separate methods for the two major sections. Each of these methods returns to the caller whenever the appropriate key has been processed, as illustrated by the method specialKeyAt:put: below. Nested conditionals are avoided.

```
keyCodeIndex < 8r200
        ifTrue:
                [self normalKeyAt: index put: value]
        ifFalse:
                [self specialKeyAt: index put: value.
                metaState ← (ctrlState bitOr: (lshiftState bitOr: rshiftState))]
```

```
specialKeyAt: index put: value
        index = CtrlKey ifTrue: [↑ctrlState ← value bitShift: 1].
        index = LshiftKey ifTrue: [↑lshiftState ← value].
        index = RshiftKey ifTrue: [↑rshiftState ← value].
        index = LockKey ifTrue: [(some series of expressions)]
```

Plan for Future Subclasses

If a method answers with a new instance expected to be like the receiver, do not use the name of the receiver´s class explicitly when creating the new instance. Use the pseudovariable self instead. For example, class Rectangle contains a method similar to the following method:

```
merge: aRectangle
        "Answer a Rectangle that contains both the receiver and aRectangle."

        ↑ Rectangle
                origin: (origin min: aRectangle origin)
                corner: (corner max: aRectangle corner)
```

Later you construct a subclass of Rectangle named Square, create an instance of the new subclass, and send the following messages:

```
large ← aSquare merge: anotherSquare.
large shrink
```

shrink is a message that only Squares can understand. The programmer assumes that the Square inherits the merge: method and will return another instance of a Square. However, the class name is explicitly mentioned in the merge: method, which therefore returns a Rectangle instead. That means large won´t understand shrink.

This pitfall can be avoided by using self and sending it the message class to create the new instance as shown below.

```
merge: aRectangle
        "Answer a Rectangle that contains both the receiver and aRectangle."

        ↑ self class
                origin: (origin min: aRectangle origin)
                corner: (corner max: aRectangle corner)
```

Organization of Classes and Methods

To modify the behavior of classes, use subclasses whenever possible, but use them wisely. A subclass should serve as a refinement of its superclass. It may define a more specific type of object, refining the semantics of a particular abstraction. For example, the classes Integer and Float both share the basic behavior of their superclass Number, but provide a more precise definition of particular kinds of Numbers. Subclassing provides another type of refinement as well: a subclass may use the same basic implementation as its superclass, but provide new abstractions. Both Set and its subclass Dictionary are implemented using hash tables, but their abstractions are unrelated.

A subclass should share some internal functional similarity with its superclass. It will inherit methods, instance variables, and class variables. As a refinement of its superclass, it may respond to additional messages or even restrict the use of certain messages with shouldNotImplement. It may also respond to the same message as its superclass but implement the method differently.

The class hierarchy is used to reflect similarity between important aspects of the *internal* implementation details of classes. Another type of grouping is provided by class categories. Classes are organized for the programmer's convenience in the Browser according to class categories. This organization reflects some *external* functional grouping.

For example, Character is a subclass of Magnitude because it inherits those protocols from Magnitude that allow its instances to be compared along a linear dimension: $a<=$b. Its class category is Collections-Text because it is used with other classes in that category dealing with text: String, Symbol, and Text. Similarly, Random is a subclass of Stream so it can inherit protocol for streaming over a collection. Its placement in the category Numeric-Numbers reflects its intended use rather than implementation details.

New Smalltalk programmers often confuse the implementation and usage aspects of organizing classes. They create a subclass so that it shares some external similarity with an existing class, and then are forced to transfer all the work to an instance or class variable. For example, they might make Random a subclass of Float, possibly creating an instance variable that would be an instance of Stream. Most methods would then deal with this instance variable. Little or nothing would be inherited from the superclass.

As well as checking to see that a new class inherits methods or variables from its superclass, think carefully about the purpose and placement of the methods within the class. A method should have a single purpose. If it performs more than one function, it should be divided into separate methods. If the method doesn't access self, instance variables, class variables or serve in an instance creation capacity, examine its placement carefully. Perhaps it belongs in a different class.

Classes are free, or nearly so. Think ahead and divide the functionality between abstract classes and refinements. Units of finer granularity allow more flexibility and greater reusability of code. Partition code so that it can be refined rather than duplicated.

Class and Instance Methods

There is a distinction between class and instance methods. Class methods serve the following functions.

- *Instance creation.*
  Methods that are used as examples of how to use a class (DisplayText example), methods

to create an instance from data in a file (Form readFrom:) , and methods used in testing (SmallInteger tests) are examples of instance creation messages.

- *Class variable access.*
  Class methods can initialize and query the value of class variables.
- *Inquiries.*
  General inquiries about information the class encapsulates are provided by many class methods. Date indexOfMonth: is an example.
- *Instance management.*
  A class may need to manage its instances, such as limiting the number or maintain a certain ordering. DisplayScreen currentDisplay: is an example.
- *Private methods.*
  Methods intended to be used only by the class itself may be listed as private methods.
- *Documentation.*
  SmallInteger guideToDivision is an example of a class method provided solely as documentation.

Instance methods do the rest of the work. They interact with the private state of a particular instance of a class. They also present a public interface to the Smalltalk programming environment.

Class and Instance Variables

Use class variables to tag constants instead of repeating them in numerous methods. Otherwise, it is easy to change a number in one method and miss it in another.

Add new class and instance variables to system classes at the end of the respective variable lists in the class definition. Not only will the standard order be preserved, but it may prevent later problems. The interpreter accesses the instance variables of certain classes by position rather than by name. If the order is changed by inserting a new variable at the beginning of the list, the system may fail. For a list of classes used by the interpreter which must not be modified, see Goldberg and Robson, Chapter 27.

Accessing instance or class variables directly may be dangerous. Send a message with the appropriate accessing protocol. It is slightly less efficient but safer and more flexible. A class will often check the integrity of its data before returning the result. It may also choose to protect its data by returning a copy of an instance variable. For example, referring directly to a window's viewport is discouraged because its value is not always current. Sending a message to ask for the viewport will guarantee that the data returned is valid.

Use class variables for shared components between all instances of a class and its subclasses. Class variables are not refinable; they are inherited and shared between the class and all of its subclasses. A subclass cannot change the value of a class variable without affecting the contents of a superclass's class variable. For values that a subclass might change, use class instance variables.

Although seldom used, class instance variables (instance variables of a metaclass) provide a similar mechanism for refining variables of a class. The values of class instance variables are not inherited or shared by any of its subclasses; each metaclass has its own unique copy of each class instance variable. If a subclass inherits the name of a class instance variable but does not initialize it, its value is nil, just as with regular instance variables.

Pool variables are used to share objects between different classes. To replace a pool variable, define a class with a class variable, then provide this class as one of the superclasses of any class that needs access to the variable.

Instance variables encapsulate information that is unique to each instance and not shared. They are refinable in the sense that each instance has its own copy. A subclass will inherit the instance variable names of its superclass, but their initial value is nil; each instance must initialize its own instance variables.

Conclusion

The rewards for writing stylistically good code include fewer errors, less time required to read and understand the code, and less frustration. The elements of good style are sometimes elusive and controversial, yet some guidelines can be identified. Some can be summarized simply.

- Learn the syntax rules.
- Avoid unnecessary white space.
- Don´t sacrifice readability.
- Follow established conventions for identifiers, capitalization, and indentation.

Others are harder to quantify.

- Each method should have a single purpose.
- Indentation should reflect the flow of control.
- Subclass wisely.
- Avoid nested conditionals.

Some of these guidelines are unique to Smalltalk; many are common to other object-oriented languages. If we agree on a common set of guidelines for writing good Smalltalk code, following them makes our task more enjoyable and benefits the Smalltalk community. It will be easier to browse, write, and debug Smalltalk code.

ADVANCED TOPICS

Multiple Inheritance of Classes

The Smalltalk-80 system supports multiple inheritance of classes which allows objects to inherit methods from two or more super classes. Ordinary subclassing only allows inheritance from a single class.

For an example of how to use multiple inheritance, go to the System Browser and choose Collections-Streams and Stream. Then choose hierarchy from the middle mouse button. In the bottom pane, you see this hierarchy:

```
Stream ()
  PositionableStream ('collection' 'position' 'readLimit' )
    ReadStream ()
    WriteStream ( 'writeLimit' )
      ReadWriteStream ()
        . . .
```

Instead of using this single inheritance scheme, ReadWriteStream could be a subclass of both ReadStream and WriteStream. Make the class NewReadWriteStream be a subclass of both by entering the following code in place of the normal class template for ReadWriteStream:

```
Class named: #NewReadWriteStream
  superclasses: 'ReadStream WriteStream'
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'Collections-Streams'
```

When you accept this code, the system generates conflicting inherited methods messages in the System Transcript because, in the case of identical messages, it does not know which methods from ReadStream or WriteStream it should inherit. You can use the Browser to eliminate the sources of these errors by specifically indicating which of the conflicting methods to use or by removing the sources of conflict.

For more information, refer to the paper by Alan H. Borning and Daniel H. H. Ingalls, "Multiple Inheritance in Smalltalk-80" , pp. 234-237, *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982.

Lazy Mutation

When you change a class definition and recompile it, existing instances of classes also must change to the new definition before they are used. This process is called mutation. Mutation involves a become: operation, a potentially expensive operation. Since not all instances of a class will continue to exist (i.e., some will be garbage-collected), the Tektronix Smalltalk system mutates only instances of a class that are referenced. It does this by *catching* messages to unmutated instances of this class and performing the mutation before actually sending the message. This mutation upon message send is known as lazy mutation. The mutation occurs only once, subsequent messages to a mutated instance operate in a normal manner. Objects marked for mutation that never receive a message are not mutated and, therefore, do not use the become: operation.

In most cases, a user will see no difference between mutating all instances immediately and mutation upon message sends. However, a certain sequence of events might lead to undesirable and unexpected consequences:

```
"redefine ExampleClass by adding a new instance variable."
all ← ExampleClass allInstances.
all do: [:each | each initializeNewInstanceVariable].
```

The mutation of an existing instance to the new definition of ExampleClass will not occur until a message is sent to the instance. At no time does this example send a message to an instance of the original ExampleClass. Instead, this code collects all the instances of the new ExampleClass (there probably aren't any) and tries to initialize them. Here is an alternative technique that will work around this difficulty:

```
all ← ExampleClass allInstances.
"redefine ExampleClass by adding a new instance variable."
all do: [:each | each initializeNewInstanceVariable].
```

In this sequence of events, all instances of the old ExampleClass are collected and *then* the definition of ExampleClass is changed. Now all the instances can be initialized and in the process mutated to the new ExampleClass definition. This technique points out that the only straightforward way to collect all instances of a class is *before* it is redefined.

IEEE Floating Point Numbers

The Tektronix Smalltalk interpreter includes primitives for many operations performed directly by the Motorola MC68881 Floating Point Coprocessor. The MC68881 conforms to IEEE-754 floating-point standards; however, Smalltalk conforms only to the representation specification of IEEE-754, not necessarily the implementation specification.

The floating point primitives are capable of generating exceptional values which print as visible Smalltalk code. These values include: positive and negative infinity, denormalized numbers, and not-a-number. Because of these new values, a new protocol for testing has been added, new instance creation methods have been added, and some existing protocol has been changed.

New methods for testing an instance of Float include testing for infinity and testing for valid numerical representation. The methods isPositiveInfinity and isNegativeInfinity return true for plus and minus infinity, respectively, while the method isInfinity returns true in either case. The method isNAN ("is Not A Number") returns true if an instance of Float does not contain a valid floating-point number representation. (Such a value is returned when dividing infinity by infinity, for instance.) isNormal returns true if an instance of Float is a valid, non-infinite floating-point value.

An instance of Float may now be instantiated to these exceptional values. Class methods negativeInfinity, positiveInfinity, and notANumber create instances of Float initialized with the appropriate exceptional value, while notANumber: creates an instance of Float in which the argument is stored in the mantissa and an all-ones bit pattern is stored in the exponent as the exceptional value.

The Float method printOn correctly interprets exceptional values by printing an evaluatable expression in each case. For example, executing print it in a workspace on 1.0e30 * 1.0e10 prints Float positiveInfinity or executing "print it" on 0.0 ln prints Float negativeInfinity.

Although the additional primitive methods represent an increase in functionality, and exceptional values are handled more completely, users of Float objects might want to protect themselves against the cascading effects of exceptional values. The Smalltalk interpreter continues evaluation using the exceptional value. The exceptional value will generally propagate through the expression, possibly making it difficult to locate the error.

Use of the new testing methods whenever there is a likelihood of generating an exceptional value is a good general coding practice.

Storing and Retrieving Objects on a File

The Smalltalk image includes a mechanism for storing and retrieving object representations (including objects with circularities) on a file (or other character stream). This mechanism has two advantages over the original Smalltalk storeOn: mechanism. First, storeOn: does not work for objects that contain circularities; second, storeOn:'s output is meant to be read in by the compiler which limits the number of literals in an object to 64. Thus, storeOn: will not correctly handle all object structures.

Using the Reading and Writing Mechanism

Four *visible* messages are defined to provide the writing or reading of objects to or from files or character-streams. The two methods for writing structures are:

someObject storeStructureOn: aStream.

Stores an object representation on a character stream aStream.

someObject storeStructureOnFile: aString.

Stores an object representation on a file named aString.

The methods for reading structures are:

Object readStructureFrom: aStream.

Answers an object defined by stream aStream.

Object readStructureFromFile: aString.

Answers an object defined on a file named aString.

Implementation Details

This implementation maps objects based on == equality. If an object has a circular structure when written out, it will be circular when read back in. Similarly, acyclic structures are read back in as acyclic structures. There are a few cautions, however:

1. There are some objects, such as processes, that may cause unexpected behavior if an attempt is made to write them out, or particularly to read back in. Contexts are treated specially in that the sender is always written out as nil. CompiledMethods are written out in a special format. Also be aware that the receiver of a MethodContext in which the block context was created is also copied as part of the definition of the MethodContext.
2. Smalltalk treats certain objects in a special way, guaranteeing their uniqueness. A new selector, isUniqueValue, has been defined that returns a boolean value, stating whether the object has this property. Such classes include UndefinedObject, Boolean, Symbol, SmallInteger and Character. Objects in these classes are mapped to the corresponding

object in the target image. Floating point numbers are written out to 9 digits of accuracy. If more (or less) accuracy is desired, it is necessary to modify the method Float printStructureOn:.

3. *This caution does not apply to objects for which* isUniqueValue *is true.* Objects that correspond to global Smalltalk names in the original image are mapped to objects with corresponding global Smalltalk names in the target image. This prevents classes and metaclasses from being duplicated. It requires, however, that you be responsible for ensuring that the target image defines all global variables that are referenced (directly or indirectly) by the object in the source image. If two Smalltalk globals refer to the same object, the result is nondeterministic.
4. Most classes are stored and read in using methods inherited from class Object, which copy instance variables and array elements using instVarAt:, etc. This means that classes must have identical definitions in both the original and target images. It also means that classes that depend on the hash values should be handled specially. Currently, only Set and its subclasses are treated specially for this reason. String and Number (and their subclasses) are treated specially for conciseness of notation (and because SmallInteger must be treated specially anyway). CompiledMethod is also treated specially.
5. A receiver's dependents (from the Smalltalk dependency mechanism) are not mapped.

Copying Circular Structures

The following methods implement a mechanism for copying Smalltalk objects that may contain circularities. The Smalltalk method shallowCopy does not generally copy the complete structure, while deepCopy only works for non-circular structures.

Two *visible* messages are defined to provide the copying of structures:

someObject structureCopy

Answers a copy of the object.

someObject structureCopyWithDict: anIdentityDictionary

Answers a copy of the object, given that a partial list of mappings from objects in the old domain to the new are in anIdentityDictionary.
The method may have side effects on anIdentityDictionary, adding new mappings.

The simplest method to use is structureCopy. However, if you want to have a handle on the mapping dictionary (either to pre-specify some mappings, to know the mappings after the copy has been created, or to get a copy of several objects that may have common subobjects), you should supply your own IdentityDictionary and use structureCopyWithDict:.

Implementation Details

This implementation maps objects based on == equality. There are a few cautions, however:

1. The copying of objects such as processes will probably cause strange behavior. When a context is copied, the sender field in the new context is nil. The receiver part of a MethodContext, however, becomes mapped to a new object just as any other object would. CompiledMethods are not copied; rather, the original object is returned. The idea here is that compiled methods should be *constant* objects.
2. Smalltalk treats certain objects in a special way, guaranteeing their uniqueness. These objects in classes such as Boolean, SmallInteger, and Character will return themselves rather than a copy.
3. Most classes are stored and read in using methods inherited from class Object, which copy instance variables and array elements using instVarAt:, etc. This means that classes that depend on the hash values should be handled specially. Currently, only Set and its subclasses are treated specially for this reason.
4. A receiver's dependents (from the Smalltalk dependency mechanism) are not mapped.

Section 6

Image and Change Management

OVERVIEW

This section presents information that helps you manage your image and the changes to that image. It also discusses how to recover your work if your image becomes corrupted and fails to work properly. Be sure you have a basic understanding of what the interpreter, image, changes, and *standardSources* files are and how they are related before you proceed with this section.

CUSTOMIZING YOUR IMAGE AND CHANGES FILE

Each time you do meaningful work in Smalltalk, you change your image. To preserve this information, each image requires its own changes file. You should go through this procedure shortly after you bring up Smalltalk for the first time. Thus, the procedure is written for the naive Smalltalk user. However, you can certainly run Smalltalk for a while before you decide you want to customize your image or changes files.

1. Activate the System Workspace window. Scroll down a short way until you reach the heading Create File System. Below this heading you see the following lines (or similar lines). Scroll until as many of these lines as possible are visible in the window at once.

```
"Make the Smalltalk home directory an absolute path"
Disk ← FileDirectory directoryNamed: (OS originalEnvironment at: #HOME).

"Make the Smalltalk home directory float to the directory in which Smalltalk was invoked"
Disk ← FileDirectory currentDirectory.

"Set pointer to Smalltalk source and changes files"
SourceFiles ← Array new: 2.
SourceFiles at: 1 put:
  (FileStream oldFileNamed: OS smalltalkInitializationDirectory  name,
  '../standardSources.Version', Smalltalk versionNumber).
SourceFiles at: 2 put:
  (Disk file: 'image.changes', Smalltalk versionNumber).
(SourceFiles at: 1) readOnly.
```

   The italicized lines are comments to the Smalltalk code. Disk is a variable global to the entire Smalltalk system. It tells Smalltalk what directory to use as its home directory. Any operations which access the external directory structure from Smalltalk use the variable Disk. Fully qualified pathnames are accepted; relative pathnames are accepted as being relative to the directory Disk is set to. When your image comes up, Disk is set to the directory from which Smalltalk was invoked.

   It also creates a file called *image.changes* in the the directory from which you invoked Smalltalk. This is your default changes file. When you save your work and exit Smalltalk, the files *image* and *image.changes* will exist in the directory from which you invoked

Smalltalk. From that point on, you can move to that directory (at the operating system level) and invoke your customized image by entering *image* instead of *smalltalk* at the command line, as you did to start this session. Entering *smalltalk* always gives you a fresh image. Entering *image* gives you your customized image. The file *image.changes* holds all changes made to your image.

2. If you wish to create a different changes file, select the file name *image.changes* and type a file name of your choice for your changes file. Don't select the quotation marks, however; leave them as they are.
3. If you have changed the file name, select the entire line reading:

   ```
   SourceFiles at: 2 put:
     (Disk file: 'image.changes', Smalltalk versionNumber).
   ```

   Press the middle mouse button down and hold it, while you scan the pop-up menu for the item do it. Roll the mouse down until this item is highlighted. Release the mouse button. You have caused your new line of Smalltalk code to be executed. Every time you execute do it, the Smalltalk interpreter executes the selected code, and the action is recorded in your changes file.
4. If your fingers slipped and you executed a different menu item instead, you can fix your mistake right now. Access the middle mouse button menu again and highlight the item undo. Then release the middle mouse button. This item undoes your previous editing action. Now select the line and try the previous step again.
5. If you access a menu, and then decide that you do not wish to execute any item on it, continue pressing the mouse button. Slide the cursor off the menu so that no item is highlighted. Now release the mouse button. The menu goes away when the mouse button is released, and because none of the items were highlighted at that time, none are executed.
6. If you wish to confine all your work in Smalltalk to a specific directory, you may wish to specify that directory to be the global variable Disk. If so, then go back up to the line that reads:

   ```
   Disk ← FileDirectory directoryNamed: (OS originalEnvironment at: #HOME).
   ```

   and select the parentheses, and everything within them. Type a single quotation mark, and the full pathname of the directory you wish to use. Close the pathname with another single quotation mark.
7. Now select the entire line. Press the middle mouse button down and hold it, while you scan the pop-up menu for the item do it. Roll the mouse down until this item is highlighted. Release the mouse button. You have caused your new line of Smalltalk code to be executed.
8. If you have made changes to the System Workspace, you probably want to save them so that any new System Workspace you access in your image contains an up-to-date version of the contents. Access the middle mouse button menu again, and highlight the command accept. You may now close the System Workspace, saving its changes.

CHANGE SETS AND THE CHANGES FILE

When you develop applications in Smalltalk or modify the basic system to suit your own requirements, you modify or add classes and methods that already exist in the system. The standard system has built-in helps that allow you to keep track of modified or additional classes and methods. The standard system also keeps track of your actions as you program.

Smalltalk preserves changes that you make to the image in two basic ways:

- It creates and continually adds to change sets.
- It continually adds to a changes file.

Change Sets

Change sets are created and managed by the class ChangeSet. Smalltalk maintains one instance of ChangeSet for each project. SystemDictionary has a class variable, SystemChanges, which is a change set for the current project.

As you make changes to classes and methods, descriptions of these changes are added to the current change set. The contents of a change set continues to grow unless you make specific modifications to it.

The following actions are recorded in change sets:

- Modifying a class definition already in the system.
- Modifying a class comment.
- Adding a new class.
- Renaming a class already in the system.
- Removing a class.
- Modifying a class´s category.
- Changing a method already in the system.
- Adding a new method.
- Removing a method.
- Modifying a method selector´s category.

Note that your actions (do its, print its, inspect its, etc.) are not recorded in a change set.

If you would like to see what your change set (in a particular project) contains, evaluate this expression in a workspace with a print it:

```
Smalltalk changes asSortedCollection
```

This collection may be rather large if you have made considerable modifications or additions to your system.

Filing out classes or methods does not affect the contents of the change set. However, by direct action you can affect the contents. For example, you can evaluate an expression to cause wholesale removal of all entries in a change set with:

```
Smalltalk noChanges
```

or, you can remove all entries that refer to a *specific* class by evaluating:

```
ClassName removeFromChanges
```

where ClassName stands for a particular class name.

If you would like to view the contents of the current change set representing modified methods, then evaluate the following expression:

```
Smalltalk browseChangedMessages
```

If there are no changed methods, then the text Nobody is printed in the System Transcript. Otherwise, a change-set browser opens. This change-set browser allows you to file out methods. You can also inquire about senders, implementors, and messages in a method too.

The Changes File

The changes file is the mechanism that Smalltalk uses to preserve changes to the programming environment. You should not use an operating system text editor, such as Emacs or Vi, to edit changes files since your image contains absolute references to code in its associated changes file.

The changes file is a super set of the change set(s) because all change set data in addition to your actions such as do it and print it are written to the changes file. However, not all actions are recorded.

If you do a do it on:

```
Pen example
```

the evaluation of this expression is recorded in the changes file as

```
Pen example!
```

However, the evaluation of some expressions and do its terminate by losing control. And, thus, are *not* recorded in the changes file. Here is an example. Evaluate the following expression using the middle button item do it:

```
4 factorial inspect
```

The new inspector takes over control before the do it can be recorded.

Changed or new methods are recorded in the changes file using the file out format. This format uses the special character exclamation (!). For example, the Pen example method would look like this:

```
!Pen class methodsFor: 'examples'!

example
  "Draws a spiral in black with a pen that is 4 pixels wide."
  "Pen example"

  | bic |
  bic ← Pen new.
  bic mask: Form black.
  bic defaultNib: 4.
  bic combinationRule: Form under.
  1 to: 50 do: [:i | bic go: i*4. bic turn: 89]! !
```

The facility to condense a changes file is provided because much of the time modifications to a method go through several iterations before a final version is decided upon. Each time the method is recompiled, it is appended to the changes file, even though only one version is referenced. In the System Workspace under the heading Changes, execution of the expression:

```
Smalltalk condenseChanges
```

renames the original changes file to a backup name and creates a new changes file with duplicates and doIts removed. Be sure to save your image after condensing changes, so that the references to the new changes file are saved.

A very important action is recorded in your changes file – the snapshot action. Whenever you make a snapshot, the following text appears in the changes file:

```
——SNAPSHOT——to <your image> <some time>
```

Thus, when you browse the changes file, you can easily determine when the latest snapshot was done.

The Change-Management Browser

A powerful way to manipulate the changes file is to use the change-management browser (also called change list view). This is described in some detail in the Addison-Wesley Goldberg book on pages 468 through 477.

The following expressions are in the System Workspace.

To create a blank view, evaluate this expression and use menu item to read a file and operate on its contents:

```
ChangeListView open
```

To create a view containing code in the file out format for a specific file, evalutate this expression:

```
ChangeListView openOn: (ChangeList new recoverFile: (FileStream
oldFileNamed: 'filename.st'))
```

PROGRAMMING METHODOLOGY

Smalltalk programmers should store applications independently of their image. This is particularly important when the development team consists of more than one person. Version control is simplified when code is independent of an image.

Here are several ways to store your code independently of your image. You can save the contents of a change set on a file by evaluating the following expressions:

```
(FileStream newFileNamed: 'filename.st') fileOutChanges
```

This expression creates a file and writes out the entire contents of the current change set in file out format. This may include new or modified classes and methods.

```
(FileStream newFileNamed: 'filename.st') fileOutChangesFor: ClassName
```

This expression creates a file and writes out the contents of the current change set referring to the specified class. (ClassName stands for a particular class name.)

The change set may include modifications not relevant to an application. You can remove changes associated with a class. The following expression removes changes associated with Stream. For example, all new Stream instance methods are removed. The record of all instance changes is removed from the current change set, but not from the programming environment. (This is an example. For your application choose an appropriate class.)

```
Stream removeFromChanges
```

The following expression removes changes associated with Stream class, a metaclass. For example, one of the kinds of changes removed is changed class methods. The record of all changed class methods is removed from the current change set, but not from the programming environment.

```
Stream class removeFromChanges
```

To remove all Stream references, you must use both of these expressions separately (in either order). The contents of the changes file are not modified by the evaluation of these expressions. However, the result of filing out your changes will not include any Stream modifications unless subsequent Stream modifications are made.

The file out changes methods produce files that can be incorporated back into an image. Use the middle button item, file in, in a file list, or execute the following expression:

```
(FileStream fileNamed: 'filename.st') fileIn
```

Or, you can also use the a ChangeListView to incorporate parts of the file.

Since a ChangeSet is an internal list of modifications, one per project, it is a useful aid to organizing application code. For instance, you might create two projects for two distinct applications in your Smalltalk-80 image. Since each project represents another screenful of information as well as another ChangeSet, this is a convenient way to organize your work. As long as changes relevant to each application are made in their respective projects, filing out changes records modifications for that project. Chapter 4 of the Goldberg book contains more information on projects.

IMAGE MAINTENANCE

Ordinarily, the Smalltalk image is stable and adding code does not decrease this stability. However, since Smalltalk allows you complete freedom to personalize and change the Smalltalk programming environment to suit your needs, there exists the possibility that you may make modifications that render your system inoperable. If you modify system level code, a good practice to follow is to alternate between two image files as you make changes to your system. That is, snapshot alternately between the two files. This way you are always able to go back to a previous image.

After you have created your own image and saved some of your work in your own image file, you may have created an image that does not load properly. If this happens, use the UTek operating system command *ls -l* to check the size of your image file. It should be at least the same size as the standard image file. The standard image file, *standardImage*, is approximately 1.3 MB in size.

CRASH RECOVERY

The changes file is output in file out format which means that you can use the file in procedure to reestablish work lost from the current image due to a crash or other malfunction. In general, you should back up to the latest good image, bring it up, and evaluate this expression:

Smalltalk recover: 5000

This expression reads the last 5000 characters of the changes file. Find the last "—— SNAPSHOT—— text and then file in expressions that you know to be good up to the crash or malfunction. If you have done a lot of work since your last snapshot, it may be necessary to recover more than 5000 characters. You may modify this expression to recover as much as you need.

Another way to recover is with the following expression. This creates a view containing just the actions and changes since the last snapshot:

```
ChangeListView recover
```

Use the middle button items do it or do all to incorporate changes into your image. ChangeListView has a number of filters that allow you to selectively view your changes. See the Goldberg book, pages 468 through 477.

CLONING

The section entitled "Creating a New System Image" beginning on page 478 of the Goldberg book discusses how and why to *clone* an image.

You can make a copy of your running image, eliminating references to inaccessible objects. This is accomplished by tracing through the running image and writing a copy of each encountered object to a file. This file contains the cloned image. Some intermediate files, suffixed with *.scratch*, are also created while cloning, but are automatically deleted after the cloning is completed.

To make a cloned image, evaluate the expression:

```
SystemTracer writeClone
```

Circular garbage is reclaimed as a part of the normal garbage collection process, so there is no need to clone an image to release these objects in Tektronix Smalltalk. You may still want to make a cloned image to release old Symbol table entries. Since a cloned image is completely re-organized, it is possible to obtain better locality of reference. This may have an effect on frequency of paging.

Besides tracing through an image to eliminate inaccessible objects, application developers may also want to incorporate the changes file into the source file. The expression to write a new source file is:

```
Smalltalk newSourceFile: 'newSourceFileName' without: Array new
```

This also creates an empty changes file, and there are no references in the image to the changes file. Incorporating the changes file causes you to lose versatility and increases the difficulty of change management, but is good for application delivery. The contents of the change sets are not modified by incorporating the contents of the changes file into the source file.

Uncollectible Garbage

In order to write a cloned image, your image must be *clean*. This means that your image must be free of uncollectible garbage, such as undeclared objects, hanging DoIt methods, obsolete classes, obsolete associations, etc. These mainly involve pointers that reference obsolete objects. You may have created uncollectible garbage by control-C'ing while filing in, by removing a class from the system while still having a class variable assigned to it, and so forth.

Although a procedure is presented here to describe how to clean up your image, you should be aware that cleaning up your image is more an art than a science. The following procedure will help you track down uncollectible garbage; however, it is not guaranteed to find all uncollectible garbage. Futhermore, once these expressions reveal the existence of some garbage, you must manually trackdown and eliminate the garbage yourself.

Many of the following Smalltalk expressions are found in the system workspace. You can execute these expression in either the system workspace or in an ordinary workspace. Be sure the system transcript is open since some of the expressions write text to the system transcript and you may want to save this text. In the following procedure, to *do* something means to select it and perform a DoIt menu operation, while to *print* something means to select it and perform a PrintIt menu operation.

1. *Do* Smalltalk forgetDoIts. This eliminates hanging "DoIts", which, for example, you may have created by doing a control-C while executing code.
2. *Do* Checker allUnscheduledDependentViews do: [:aView | aView release]. You may have some unscheduled windows that are still referenced via the dependency mechanism. This releases unscheduled windows.
3. *Print* (Object classPool at: #DependentFields) keys. This prints a set of object dependents, some of which may be garbage. There should typically be an object for each open view, including those in other projects. Typical ones are TextCollector (system transcript), an InfiniteForm (the background), various workspaces, various browsers, etc. If you have garbage dependents, execute the next code in the system workspace with the

argument to isKindOf: the appropriate class to release only the garbage desired. This expression may have to be executed more than once.

4. *Do* Undeclared inspect. The resultant inspector should normally be empty. If not, check for references and remove or declare as appropriate.
5. *Print* Checker obsoleteClasses. This should result in an empty OrderedCollection. If not, use Smalltalk collectPointersTo: to eliminate the references to each obsolete class. Make sure you save the result. You may also want to use Checker obsoleteAssociations in the same manner to help eliminate obsolete classes.
6. *Do* Checker rehashBadSets. This verifies that keys for sets are valid. The system transcript prints a message saying how many sets had to be rehashed.
7. *Print* the following code:

```
Smalltalk classNames select:
  [:x| (Smalltalk at: x) superclass
  class ~~(Smalltalk at: x) class superclass]
```

8. *Print* the following code:

```
"Check for missing classes in subclass lists."
Smalltalk allBehaviorsDo: [:class | sClass ← class superclass.
  sClass notNil ifTrue: [(sClass subclasses includes: class)
    ifFalse: [Transcript show: class  printString,
    ' is missing from superclass ', sClass printString]]].
```

9. *Print* the following code:

```
"Check for duplicate or erroneous classes in subclass lists."
Smalltalk allBehaviorsDo: [:class | subs ← class subclasses.
  subs do: [:each |
    (each superclass ~~ class)
      ifTrue: [Transcript show: each printString,
      ' is incorrectly duplicated in subclass list of ',
      class printString; cr.  class removeSubclass: each]]].
```

10. *Print* the following code:

```
"Check for classes in subclass lists which are not contained
 in the system dictionary."
Smalltalk  allBehaviorsDo: [:class |  class subclasses do: [:each |
  ((Smalltalk at: each name ifAbsent: [nil]) = nil and: [each isMeta not])
    ifTrue: [class removeSubclass: each.
      Transcript show: 'Removing subclass ', each printString.
      ' from ', class printString; cr]]
```

Classes that have incomplete or incorrect references within the class hierarchy either are not written in the clone image or cause the tracer process to break. Before using the

SystemTracer, fix the class hierarchy in the image based on information from steps 8, 9, and 10.

Section 7

MODEL-VIEW-CONTROLLER

INTRODUCTION

Objectives

After reading this section and working the example you should have a good understanding of the Model-View-Controller paradigm. You will learn the ways in which these objects interact, and how to make a simple application using a Model-View-Controller triad.

Overview

As a new Smalltalk programmer, you´ll eventually face the moment of truth. You´ve got a good idea of the way an object-oriented language works, the object/message-passing paradigm has begun to make sense, and you´ve realized that the syntax of Smalltalk is really not that different from other languages. You´ve even got an idea for the first program to write. You know how you want your data structured, you know what algorithms you need. Now come the questions: "Just how do I present this to the user?" and "How do I get the user´s input?" What begins as a simple problem becomes an exercise in writing menus and creating an output format. The joy of creation can become lost in the demands of making a simple application robust and user-friendly.

Tektronix Smalltalk contains a large set of tools: classes to make the task of programming a user interface easier. You´re not required to use these tools — Smalltalk is, after all, a general-purpose programming language — but it´s more consistent (and much easier) to use them.

To see how applications work in the Smalltalk environment, spend some time browsing through existing applications. Look at the standard Smalltalk environment itself, the demonstrations in the executable file */usr/lib/smalltalk/demo/demoImage*, and the applications in the */usr/lib/smalltalk/fileIn* directory and you´ll find that these applications share a common "feel." A typical Smalltalk application runs in a window that becomes active when the cursor enters it. You select things or give graphic input using the mouse (and sometimes the left mouse button). You select tasks from context-sensitive menus that pop up when you press the middle mouse button. You manipulate the window in which the application runs by making selections from a menu that pops up when you press the right mouse button. You select items from menus holding the mouse button held down, selections are highlighted when you move the cursor over them, and the system executes the highlighted selection when you release the mouse button.

If you make your application follow the Smalltalk conventions, someone using it for the first time won´t have to learn an entirely new way of interacting. Tektronix Smalltalk contains the pre-written code to let you establish windows, sense the position of the cursor and the condition of the mouse buttons, create pop-up menus, and use the placement of the cursor to take and pass control like existing applications.

One key to writing Smalltalk applications is organizing the task. Smalltalk typically divides an application into three main parts:

- The model. An object that contains the data and algorithms that you manipulate to solve a problem.

- The view. An object that displays information in a window to the user.
- The controller. An object that lets the user interact with your model.

Almost all interactive features and applications in Smalltalk, such as browsers, workspaces, and file lists, are divided into these functions. This separation is so often used that it is known as the Model-View-Controller paradigm (often abbreviated MVC). The relationship between the objects in the MVC paradigm is extremely tight; the mechanics of the MVC is not readily apparent to a new Smalltalk programmer. This discussion and the following example clarify the interactions between models, views, and controllers.

WHAT IS AN INTERACTIVE APPLICATION?

Any computer application has three main functions. These are:

Input	An application provides some means for the user to interact with it.
Process	The application manipulates and stores data.
Output	The application gives the user some feedback about the process.

Two differences between an interactive and a batch application are the user´s perception of the application and the scope, or quantization, of control. In a batch application, you enter your commands, then wait an for an output. In an interactive application, you get immediate (or nearly so) feedback, then enter more commands and get more feedback, and so forth.

The scope and quantization of control also differ between batch and interactive applications. Batch commands are generally few and far-reaching. Interactive commands, on the other hand, generally do less. You must repeat commands more often. In short, a successful interactive application gives you the feeling of continuously controlling the computer while the batch application feels more like the instructions on old Chinese fireworks: "Light the touchpaper and retire."

MODEL, VIEW, AND CONTROLLER

Smalltalk divides applications into models, views, and controllers. The controller accepts input. The model performs the processing. The view displays the output.

Model

A Smalltalk model is the processing part of an application. It stores and processes data. Object, the fundamental class in Smallltalk, contains all the necessary behavior to be a model. All objects in Smalltalk inherit from the class Object, therefore, *any Smalltalk object can be a model.*

Models need not contain information about displaying themselves or interacting with a user.

View

A view is Smalltalk's most common output mechanism. It displays the model (or some aspect of it) to the user. A view can be any representation of the model such as:

- a map
- a schematic diagram
- a paragraph of text
- a list of relationships
- a bar chart
- a graph

or any other way of representing information on the screen. In addition, a view maintains its own coordinate system so you need not scale graphics to fit the view's window — it handles graphics conversions for you. Views can contain other views (called *subViews*) to show more than one aspect of a model.

Smalltalk views are almost always subclasses of the class View.

Smalltalk views are tightly coupled with controllers; each view is coupled with a single controller. A controller usually takes control only if the cursor is inside the view.

A model's only obligation to its view or views is to make it known if it changes. A view is dependent on and must query its model to display it, but a model normally knows nothing of its view or views. Indeed, one model often has multiple views showing different aspects of the model.

Controller

A controller is a Smalltalk input mechanism. A controller interprets mouse and keyboard activity when it has control of the system. A controller is normally active, or takes control, while the cursor is within its view.

Each window visible on the Smalltalk screen has an associated controller. When you move the mouse into a window, the controller for that view (or subview) becomes active and interprets your input in the context of the view. For example, pressing the middle mouse button in a System browser window gives you a different menu than pressing the middle mouse button in a workspace window.

The Smalltalk system cycles continuously through a loop checking whether controllers want control. A system controller manager asks each controller on a list of scheduled controllers, in turn, if it wants control. Controllers take control if the cursor is within its view, perform their control activity, then relinquish control.

Smalltalk controllers are usually subclasses of the class Controller.

THE Model-View-Controller TRIAD

A Simple Mvc Triad

One model, one view, and one controller working together form the simplest possible Smalltalk application, a Model-View-Controller triad. Figure 7-1 shows a simplified illustration of how the Model-View-Controller triad works.

1. The Controller accepts your input, and causes the Model to change.
2. The Model informs the View that it has changed.
3. The View then displays information about the Model in a window on the screen.

Later in this section, you will find a working example of a MVC triad that you can type into your system and run. If you find yourself becoming confused as you read through this discussion, remember that you can look ahead and read the working code. You must look at models, views, and controllers from the perspective that they form a system and interact with one another.

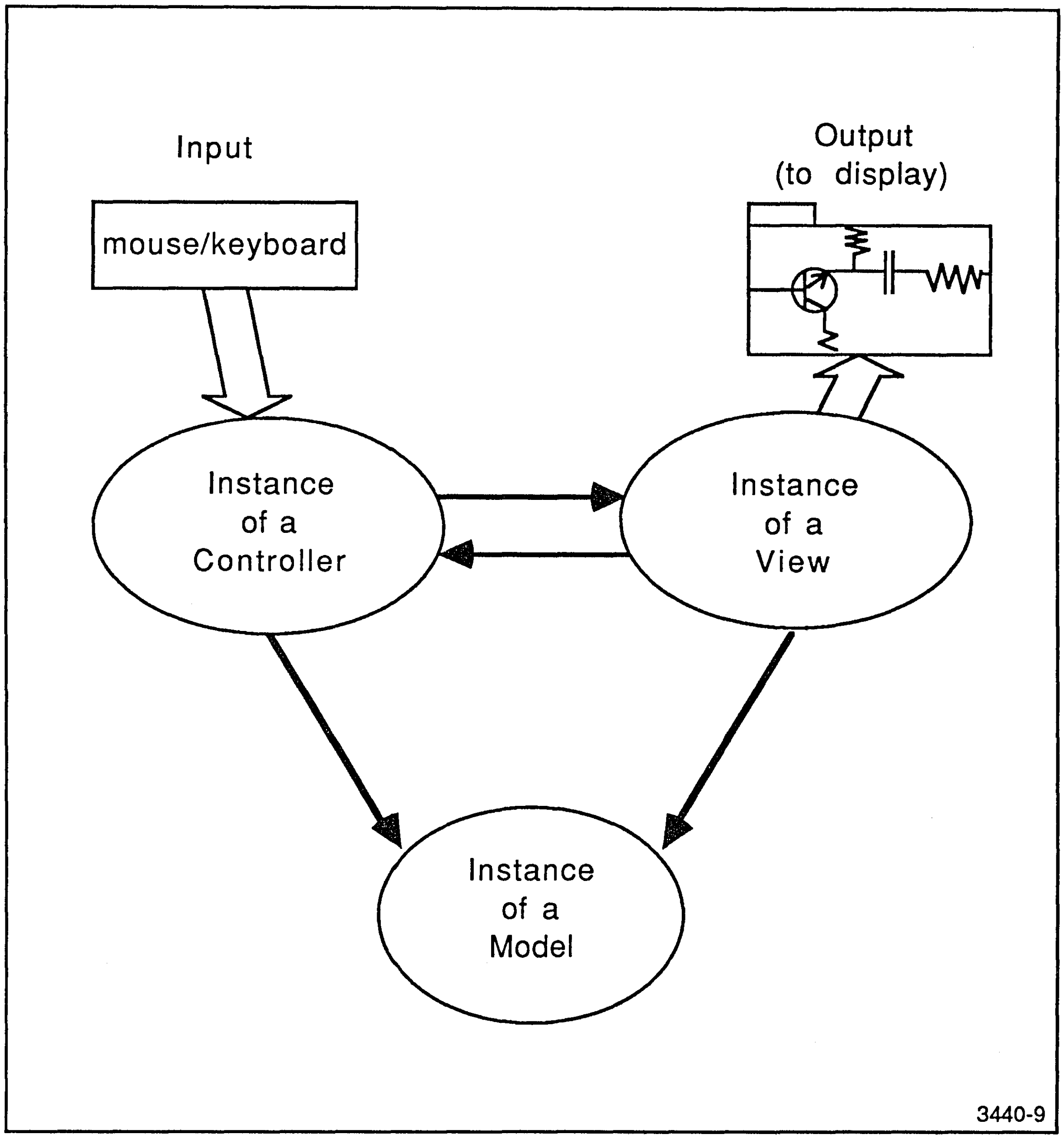


Figure 7-1. A Model-View-Controller Triad.

In this triad, the model is nearly independent of the other two (we'll discuss the specific linkage later), but the view and controller are tightly coupled to each other and depend on the model. In most applications, you'll develop the model first, then link it to a view and controller. Assuming that the model has been developed, let's look at how views and controllers communicate and behave, then see what a model has to do to become part of the triad.

View Behavior

A view´s behavior depends on:

- The behavior it inherits from the class View.
- The behavior the Smalltalk programmer gives it.
- Its relationship to its subviews and whether it is a subview of another view.

Subviews

Views can contain other views. Any view that is contained within another is called a subview of the view that contains it. Although you can nest subviews within each other, this nesting is rarely more than one deep; all subviews are usually subviews of a common view.

For most applications you will make your views subviews of a specialized view (usually called the *topView*). The topView is an instance of a StandardSystemView (another subclass of View). Thus, the overall view of a model is normally a topView that has one or more views as subviews of the topView. Figure 7-2 shows how several views may be used to make a complex view of a model.

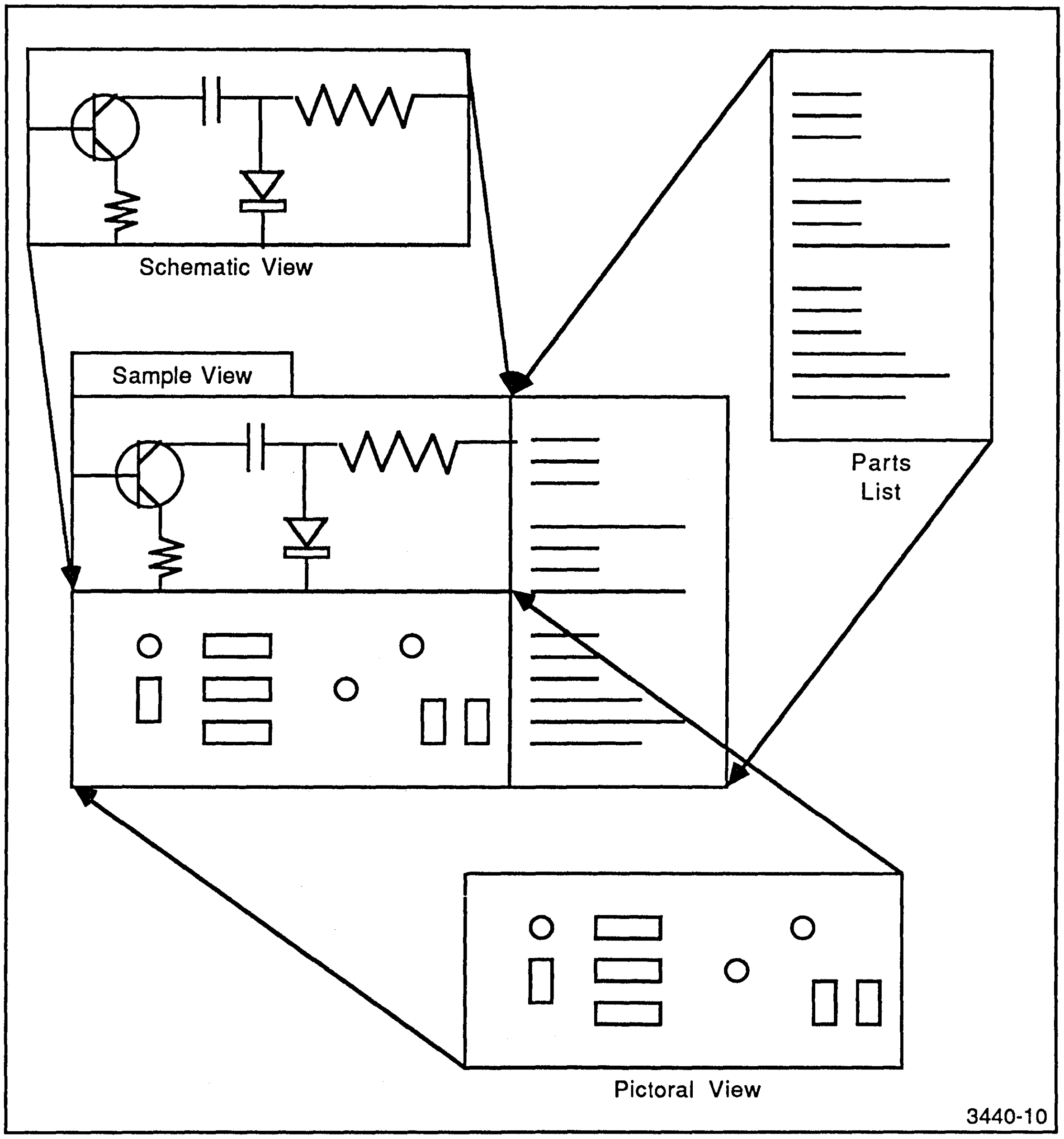


Figure 7-2. Combining Subviews for a Complex View.

View Inheritance

You create new classes of views to display some aspect of your model. You normally subclass these new classes from the class View so instances of these new classes inherit the behavior and instance variables of class View; they can display their interior, their borders, and their subviews. In addition, each inherits instance variables that specifies its model, controller, and the list of its subviews. You must write the methods that display the particular aspect of the model you want to show.

Using Views in Smalltalk

In order to conform to the Smalltalk environment, you normally make all your views (even if you only have one) subviews of an instance of StandardSystemView. StandardSystemViews give you the familiar title tabs and have their own controllers, instances of StandardSystemController.

The controller for a StandardSystemView determines what the system does when you press the right mouse button. Therefore, making the topView a StandardSystemView gives you, for little overhead, a window that conforms to the Smalltalk conventions for windows and the right mouse button.

A StandardSystemView's model may be either your model or nil. If you browse through examples in the system, you'll find some examples where the model for the SSV is the underlying model and some examples for which the model is nil. The only existing situation in which this is important is when you are working with text views. If you want the conformation window to protect your text (pop-up a window to ask if you really want to close), you must make the SSV's model the model of the text view.

Figure 7-3 shows how a MVC triad relates to a StandardSystemView and StandardSystemController.

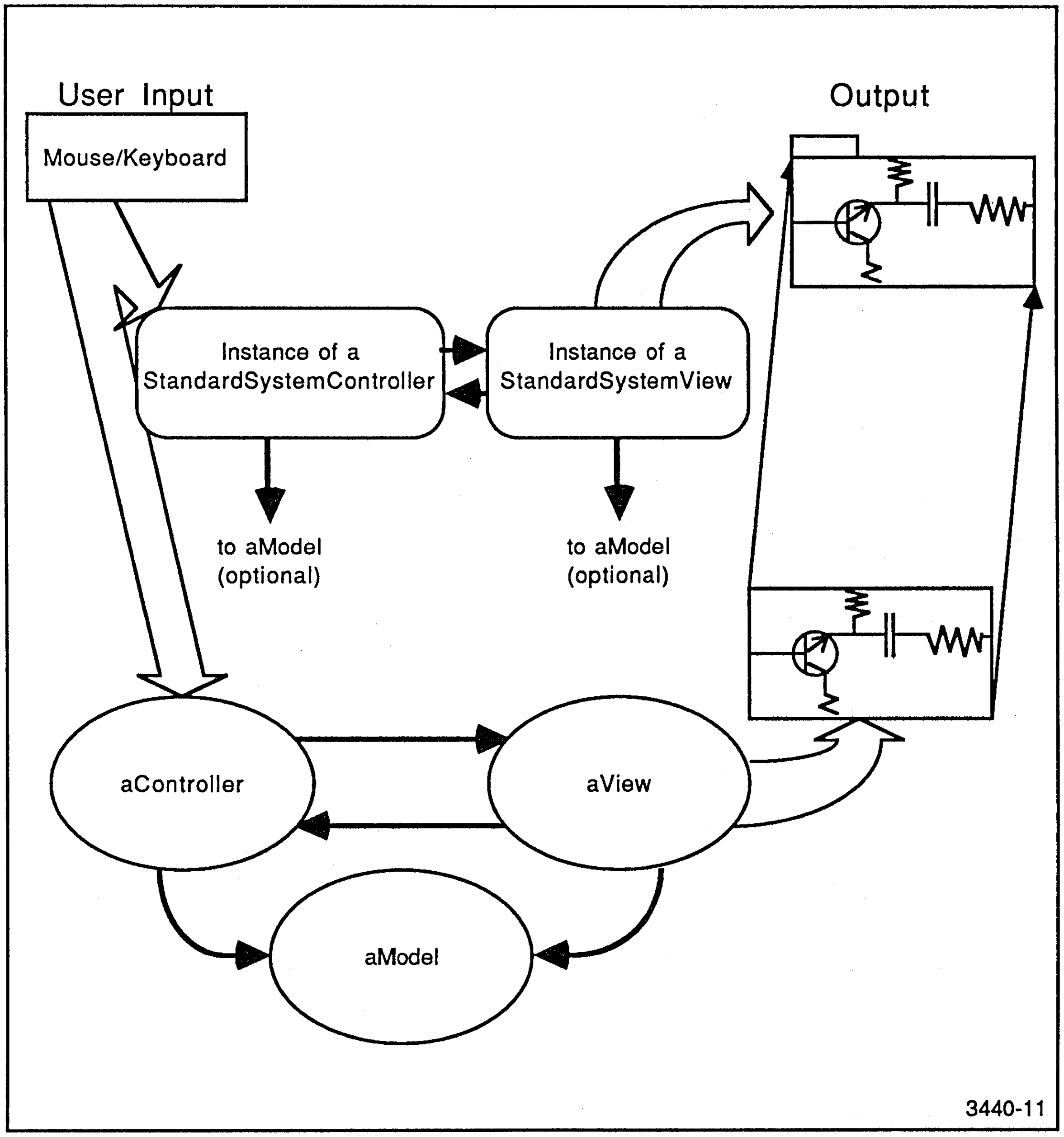


Figure 7-3. MVC Triad with StandardSystemView and StandardSystemController.

How Views are Displayed

In the class View, the method ´display´ determines how each View will display itself. Part of the code reads as follows:

```
self displayBorder
self displayView
self displaySubViews
```

When an instance of View or one of its subclasses gets the message display, it first displays its own border, then tells itself to execute the method displayView. (The method displayView is the method you must write for each of your Views to display its model´s information.) Finally, the method displaySubViews sends the message display to each of the View´s subviews. Thus, when we send topView the message display, the display message is propagated down the display hierarchy.

Creating A New Mvc Triad

When you create a new MVC triad, you usually start with an existing instance of a model. To create a MVC triad you must create an instance of your view, set its instance variables and create its display hierarchy, then create an instance of a controller and set its instance variables. You can, of course, do this manually, one step at a time. However, you´ll usually want a method to do the dirty work for you.

The Smalltalk convention is to have a class method for your view (often called open or openOn:(aModel)) create a new view, set the new view as a subview of StandardSystemView, and open the StandardSystemController associated with the StandardSystemView. The system then queries the newly created instance of your view (via the method you write called defaultControllerClass) and sets the newly created controller´s instance variables.

Assuming that you´ve created the classes for your model (class MyModel), your view (class MyView) and your controller (class MyController), you might create an instance of your model by executing in a workspace:

```
aMyModel <- MyModel new
```

Now, since you´ve already written the openOn: (aModel) class method and defaultControllerClass instance method, all you need do to create the triad is execute the following method in a workspace:

```
MyView openOn: aMyModel
```

Class MyView then creates an instance of MyView with the topView a StandardSystemView and its model set to the instance of MyModel named aMyModel. In addition, the new instance of MyView is registered as a dependent of aMyModel. The system then queries the new instance of MyView (via defaultControllerClass) then creates a new instance of MyController. The system sets the view´s instance variable, controller, to the new instance of MyController, the controllers instance variable, view, to the instance of MyView, and the controller´s instance variable, model, to aMyModel. Figure 7-4 shows the sequence of events that create the view and controller and link them to the model to create a MVC triad.

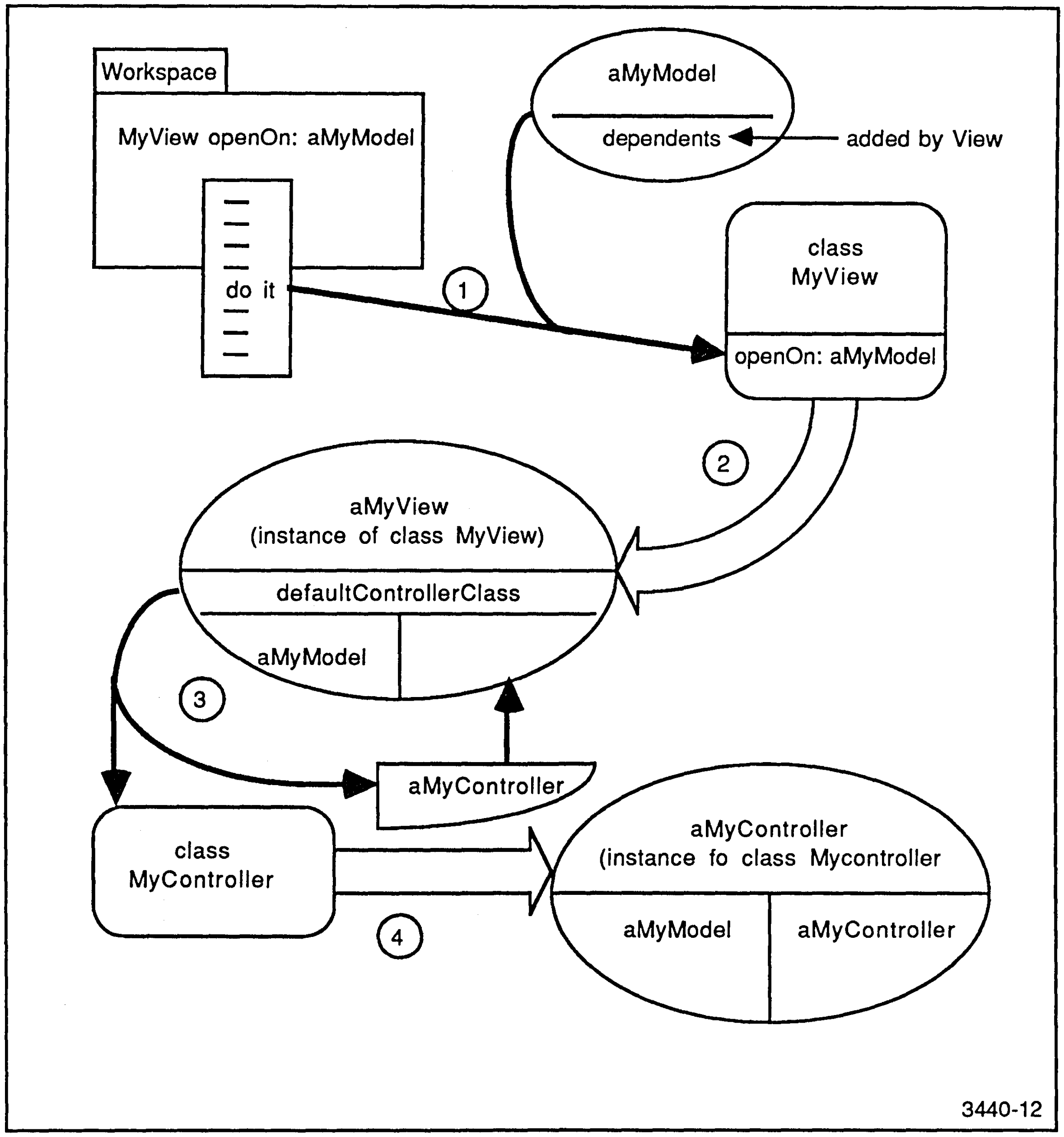


Figure 7-4. MVC Creation Sequence.

Controller Behavior

Controllers, like views, store their information about the rest of the triad as instance variables. Typically, an instance of a view creates its controller when created, leaving you only to decide what the new controller is to do. You must write the method named controlActivity to tell your controller how to respond to user input.

The basic sequence of messages that a controller receives is:

```
controlInitialize
controlLoop
        isControlActive
        controlActivity
controlTerminate
```

If you make your controllers subclasses of the class Controller, all methods except for controlActivity are already defined for you.

Should you require a completely passive view — one that you cannot affect— you can use the predefined controller NoController. This controller does nothing interactive; it simply exists. If you don't write a defaultControllerClass message for your view, you'll get an instance of NoController by default,

The Role Of The Model

The model responds to messages the Controller or other objects sends to it and does whatever it is designed to do. A model's only other task is to inform its dependents (views of it) whenever it changes.

Each object (remember, the model is a subclass of the class Object) maintains a list of dependents. Whenever the object receives the message changed:(aParameter) or changed, it sends the message update:(aParameter) or update to each of its dependents. Since Views are dependents of their models, they must also respond to either the update:(aParameter) or update message from their models (usually self display). Thus, when the model changes either as a result of its own processing or as a result of a message, it sends the message that it has changed to itself; that message, in turn, causes the model to send each of its dependents the message causing themselves to update themselves. The view, a dependent, then redisplays itself to show changes in the model.

If you want some views to respond only when one particular aspect of the model has changed, use the update:(aParameter) message, while a general change can be handled with the update message. Remember, Smalltalk takes care of the details for you. You must remember to send the changed:(aParameter) or changed message when the model changes and include the update:(aParameter) or update message in the view in order to use this mechanism. Figure 7-5 shows in more detail the relationship between the members of a working MVC triad.

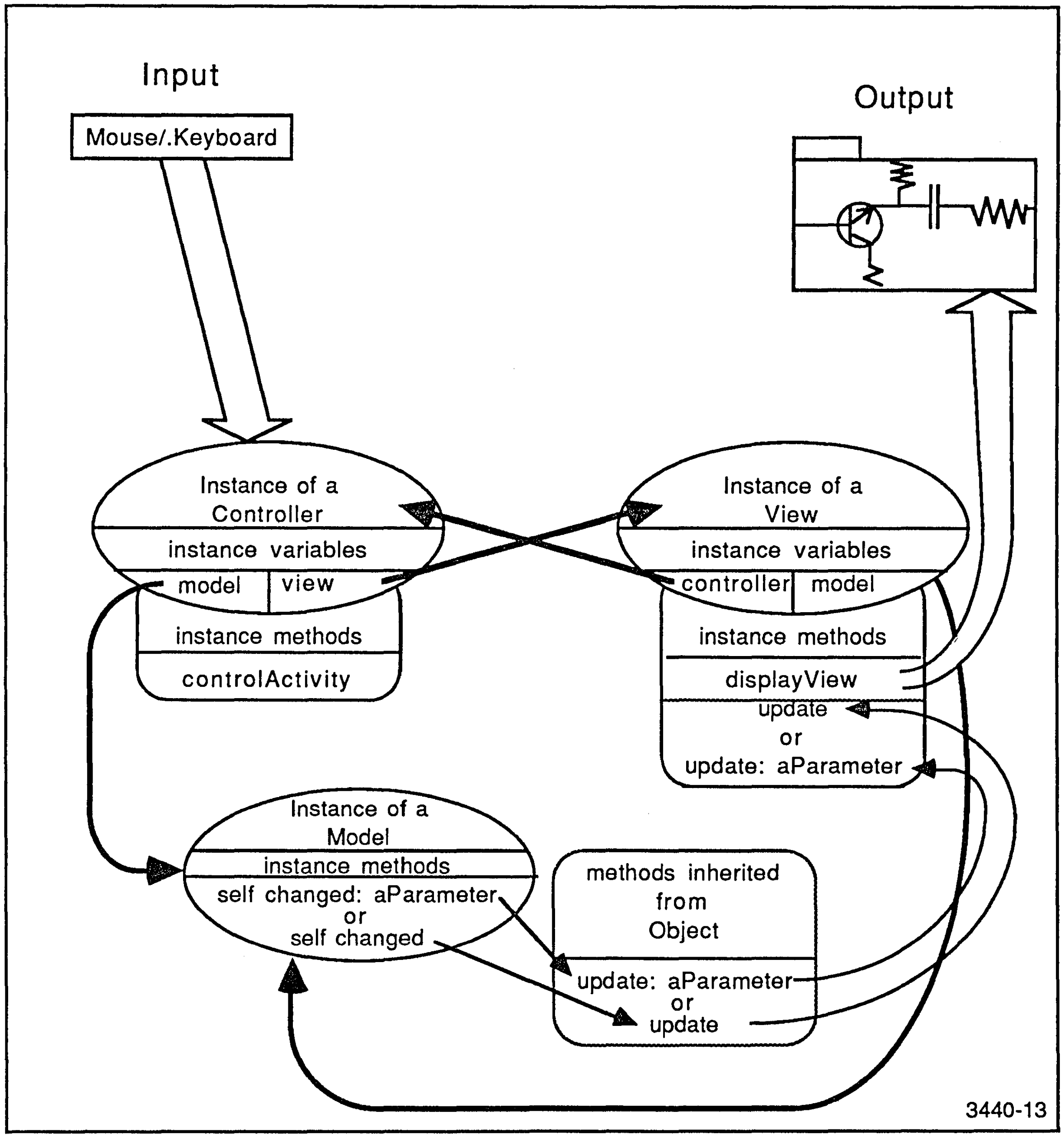


Figure 7-5. Model-View-Controller Relationship.

MVC SUMMARY

You must follow certain conventions to take advantage Smalltalk's support for the MVC paradigm. Some methods you write are used for creating new instances of the MVC triad and others are used in a running system.

Model Summary

The Model's Methods in a Running MVC Triad

A model's only responsibility is to broadcast the notification that it has changed when it changes. To do so, every method that changes the model must include the line self changed. If you write new methods that change the model, be sure that the line self changed is executed each time the model changes. If you use inherited methods, you should override them with the sequence:

super (inherited method).

self changed

If you have views looking at different aspects of the model, the model can send a parameter indicating that a particular aspect has changed with the message changed: (aParameter), where aParameter indicates the aspect of the model that has changed.

Creating a Model For a MVC Triad

You will normally create the model for a MVC triad independently of the view and controller. A model normally knows nothing of its view and controller.

View Summary

A view presents information about the model to the user. Views also create the MVC triad.

The View's Methods in a Running MVC Triad

For a view to operate in a MVC triad, you must write the instance methods that display the view and the methods to respond when the model changes. To display the view, you must write a method with the name displayView that tells your instance of a view just how to display itself. To respond to model changes, you must write the instance method named either update or update:. If the model sends the message changed, the view must respond to the message update, while if the model sends changed: (aParameter), the view must respond to the message update: and take action if the parameter is one that it should respond to. In most cases the view will simply execute self display when the model changes.

Methods to Create New Views and MVC Triads

You must write the class methods that create new instances of your view. This is often called openOn: and takes your model as a parameter (however, the name is not important). In addition, the instance creation message usually makes your view a subview of a StandardSystemView, sets the minimum and maximum sizes of your view, the border width, and inside color. In addition, you must write the instance method defaultControllerClass that determines what controller will be assigned to your new instance of your view (if you don't write this message, an instance of NoController will be assigned). When you invoke the instance creation message, the methods inherited from class View create the controller and link the model, view, and controller into a MVC triad.

A MVC EXAMPLE

Although the Model-View-Controller paradigm is conceptually simple, its implementation sometimes seems obscure. This example shows a simple model (nearly all its behavior is inherited) tied to a simple view and a simple controller. This example develops a MVC triad using existing code, then shows how to add behavior to this triad to make a very simple interactive application.

This example illustrates the relationship between a model, a view, and a controller — not the details of any algorithm within a model. Therefore, this example's model is a subclass of OrderedCollection and adds only those methods necessary to make it work with a view and controller. The model is an OrderedCollection of Points. (OrderedCollections don't care what they are collections of, you choose that by the way you use them.) The model is subclassed from OrderedCollection to inherit OrderedCollection's methods; the example overrides these methods or adds new methods only where necessary.

You can use any class in Smalltalk-80 as a model because the fundamental behaviors needed for a model are found in the root class, Object. Every Smalltalk-80 object is a subclass of Object and has all the necessary behavior to be a model. OrderedCollections are common starting points for many applications and their behavior is interesting. Points are easy to relate to graphic shapes and positions. Therefore an OrderedCollection of Points is a useful model for this example.

Enter this example into your system as you follow along. Be sure you understand what's happening at each step, and feel free to experiment.

Creating A Model Class

To create your graphic model class (ExampleModel) add a Category, MVC-Example, in the Category pane of the system browser. (To do so, move the mouse to the top left, or category, pane of the browser and select add category from the left button menu. Type the category name in the notifier, then accept it.) When the template appears in the text pane of the browser, edit the template to read as follows, then accept' it via the middle button menu.

```
OrderedCollection variableSubclass: #ExampleModel
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC-Example'
```

Once accepted, you see the new class appear in the browser. The new class has all the behavior necessary to act as a model; in fact any subclass of Object (and hence any Smalltalk object) inherits these necessary behaviors from the class Object. You can make use of these inherited behaviors to make a new instance of ExampleModel, add and delete things to and from it, and inspect it with an inspector. (To see the inheritance hierarchy, move to the class pane of the browser and select the middle button menu item hierarchy on the class ExampleModel.)

Take a few moments to get familiar with your new class. Open a workspace, and execute the following code:

```
aModel ← ExampleModel new.
aModel inspect
```

You can modify the model via methods inherited from OrderedCollection. You can add objects with add: and delete objects with removeFirst and removeLast. Leave the inspector open (or open another) and execute the following lines in the workspace. Check the object via the inspector after every line. (Remember that you must reselect self, or the field you are examining, each time you change the object before the inspector will show the change.) Try executing lines more than once. How does the OrderedCollection handle multiple occurrences of a member of the collection? How does an OrderedCollection deal with different numbers of objects? What happens if you try to remove something that isn't there? What happens to the inspector if you add more than 10 elements? Open another inspector (if you haven't erased the old code, you can select and do it again).

```
aModel add: 10@10.
aModel add: 20@20.
aModel add: 10@15.
aModel add: 100@410.
aModel add: 50@106.
aModel removeFirst.
aModel removeLast.
```

You should now be comfortable with the behavior of your model, an OrderedCollection. Don't close the workspace, you can use it later.

Class Comments

Each class you add to your Smalltalk image should include a meaningful class comment. To add a comment to a class, select the class ExampleModel in the class pane of the browser, make sure the instance switch is set on (it should be highlighted, if it isn't move the cursor to it and click the left mouse button), and select the middle button menu item comment. You should see the comment "This class has no comment" in the text pane. Edit the comment as follows and accept it.

```
————————————————————————————————————————
This an example model that has no unique behavior of its own.  All added
behavior is to work with a ExampleController and a ExampleView.
————————————————————————————————————————
```

Creating A View On The Model

Now that you have a working model, create a view on this model. You should make it a subclass of View. In the Browser, deselect ExampleModel (click on it). When the template appears, edit the template to appear as follows, then accept it.

```
————————————————————————————————————————
View subclass: #ExampleView
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC-Example'
————————————————————————————————————————
```

Don't forget to add the class comment!

```
————————————————————————————————————————
This is a simple view on a ExampleModel.  It assumes that the model is an
OrderedCollection of Points.  It does no error checking and will break if you
stuff anything else in the model.
————————————————————————————————————————
```

The view is the most complex part of a MVC triad. You send the class (ExampleView in this case) a message that causes it to create an instance of its class, ties it to a model, gets the controller type from the model, creates a controller, ties the new instance of the controller to the view, and ties the controller to the model. In addition, it creates a StandardSystemView and sets the newly created view as a subview of the StandardSystemView. Furthermore, it takes care of housekeeping such as border width, interior color, and the text in the title tab.

Instance creation is a class task — a class creates a new instance of itself. In the class pane of the Browser, select and click on the class switch. Move to the protocol pane of the browser and select the middle button menu selection (yes, it's the only one) add protocol. When the "fill in the blank" window appears, type instance creation and either accept or press the carriage return key. A new template then appears in the text pane.

You're now ready to create the method the class ExampleView will use to create new instances of ExampleView. Call this method openOn: as you'll want to use it in the form:

```
anExampleView openOn: anExampleModel
```

Edit the template to appear as follows, then accept it.

```
openOn: anExampleModel
        "Creates a new ExampleView on anExampleModel."

        | anExampleView topView |
        anExampleView ← self new.
        anExampleView model: anExampleModel.
        anExampleView insideColor: Form white.
        topView ← StandardSystemView new.
        topView label: 'Graphic MVC Example'.
        topView borderWidth: 1.
        topView addSubView: anExampleView
        topView controller open
```

Let's look at this code and see what it does:

```
openOn: anExampleModel
        "Creates a new ExampleView on anExampleModel."
```

These lines name the method (openOn:) and establish that it will respond to a keyword message. It expects the invoker to pass it the name of a ExampleModel that openOn: will use internally. In the method, you use anExampleModel to designate this passed ExampleModel name. The comment in quotes merely tell the purpose of this method.

```
        anExampleView ← self new.
```

This line creates a new instance of anExampleView. The remainder of the method initializes this newly created ExampleView.

```
        anExampleView model: anExampleModel.
        anExampleView insideColor: Form white.
```

Here you tell your new ExampleView, anExampleView, that its model is identified by the argument after the openOn: message. The second line sets its inside color to white.

Typically, you like all the windows in your environment to behave in the same way. You like to have a title tab and be able to manipulate them with the right mouse button menu. The easiest (and potentially easiest to modify and expand) way to get these functions into your View is to make your View a subView of a StandardSystemView.

The next lines of code create a new instance of a StandardSystemView and make the View you just designed a subview of the StandardSystemView.

```
topView ← StandardSystemView new.
topView label: 'Graphic MVC Example'.
topView borderWidth: 1.
topView addSubView: anExampleView.
topView controller open
```

The first line creates a new StandardSystemView and assigns it to the name "topView." The second line puts the label "Graphic MVC Example" onto the title tab of the View. The third line sets the width of the topView's border to one pixel, while the fourth makes your ExampleView a subview of the new StandardSystemView.

The final line activates the topView's Controller, already created by the StandardSystemView so that you can interact with your demonstration window. (The actual sequence is more complicated, but Smalltalk takes care of the messy details for you.) If you do nothing else, your window will now display and respond to the right mouse button menu.

ExampleView Instance Behavior

After you have created an instance of a ExampleView, you want that instance to display its model. In the class pane of the system Browser, toggle the instance switch (put the cursor on the box marked "instance" and click the left button), and, in the protocol pane, add the protocols displaying and controller access.

Select the protocol displaying, edit the template as follows, then accept it.

```
displayView
        "Display the model as black dots at each point."

        model do: [:point | (Form dotOfSize: 5)
                displayAt: (self displayTransform: point)]
```

The code is simple — it loops through the members of the model (do: is inherited from OrderedCollection) and displays a dot at the location of each point. One point to note, is that you use the View method displayTransform to change the coordinates of each point from screen coordinates to window coordinates for the View. Remember that a view sends itself the message displayView in response to the message display.

Select the protocol controller access, edit the template as follows, then accept it.

```
defaultControllerClass
        "Answer the class of the default controller for the receiver."

        ↑ ExampleController
```

Since you haven't yet created the class ExampleController, the compiler complains. Select undeclared and the method will compile. This method completes tying the ExampleView to its model and Controller — the model is declared when you create the ExampleView, the Controller is declared by the instance method defaultControllerClass.

Creating the Controller for the ExampleView

To create a Controller for the ExampleView, deselect the class in the class window of the Browser, edit the template as follows, then accept it.

```
Controller subclass: #ExampleController
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MVC-Example'
```

The class comment is:

```
I am a Controller for a demonstration Model-View-Controller triad. I do
nothing.
```

There is one required instance message selector for a Controller, controlActivity. Add the protocol control defaults, edit the template as follows, then accept it.

```
controlActivity
        "This controller does nothing"
```

While not yet complete, you have enough of the MVC triad linkage to test and see that it works. In your workspace, enter the following and do it.

```
aView ← ExampleView openOn: aModel.
```

Now frame a window just as you do any Smalltalk-80 window — its contents will be the graphic display of your model. Return to your workspace and add or delete points from the model. Nothing changes. Why? Of course the view doesn't change, because the model doesn't inform the View that anything is different. (You can verify that the changes are real with the right button menu selection 'repaint.')

Getting the Model to Inform the View of Changes

If you will recall, earlier in this discussion it was pointed out that a model has the responsibility of informing its dependents of any changes. You've made changes via the inherited methods add: and removeFirst and removeLast. To get the model to inform its dependents that it has changed you will override these methods. The overriding methods pass the original message to super, then inform the models dependents of the change with the message self changed.

Select the class ExampleModel and in the protocol pane add the instance protocols adding and removing. enter and accept the following two methods in the protocol for removing.

```
-------------------------------------------
removeFirst
        "Execute the inherited method, then inform dependents of a change."
        super removeFirst.
        self changed

-------------------------------------------

-------------------------------------------
removeLast
        "Execute the inherited method, then inform dependents of a change."
        super removeLast.
        self changed

-------------------------------------------
```

Then, under the protocol adding, enter and accept the following:

```
-------------------------------------------
add: aPoint
        "Execute the inherited method, then inform dependents of a change."
        super add: aPoint.
        self changed

-------------------------------------------
```

Now test the code. Add a point or two in the workspace. Does the update happen as soon as you do it in the workspace? Why not? When the model informs its dependents of a change it sends the update: message to each of its dependents. Your View (a dependent of your model) needs to respond to this message. Add the protocol updating to the instance methods for ExampleView, edit the template as follows and accept it.

```
-------------------------------------------
update: aModel
        "The model has changed.  Redisplay the view."
        self display
-------------------------------------------
```

Move onto your example window and activate it (you may have to click the left button). What happens then? Add and subtract a few points until you feel that you understand what's happening.

Adding Some Controller Activity

Although you now have a complete Model-View-Controller triad that works together, it feels incomplete — as though something were missing. And, of course, it is — the Controller does nothing. It's annoying to have to use a workspace to manipulate the model.

Let's complete this example by having your Controller interact with the View and model. In this simple example, you won't develop the "philosophically correct" Smalltalk-80 interface (one in which the left button would be used for selection/graphic input and the middle button to pop up a menu of choices). Instead, you'll just use the left button to add a point (within the View) to your model and the middle button to remove the oldest point from the model. To do so, you need to modify the ExampleController method controlActivity. Select it, edit it to read as follows, then accept it.

```
-------------------------------------------------
controlActivity
        "This Controller does some simple editing"
        Sensor redButtonPressed ifTrue:
                [model add: (view inverseDisplayTransform: Sensor cursorPoint)].
        Sensor yellowButtonPressed ifTrue:
                [model removeFirst]
-------------------------------------------------
```

The logic of this method is simple. If the left button is pressed (Sensor redButtonPressed ifTrue:) it executes the first block and directs the model to add the point returned by the expression view inverseDisplayTransform: Sensor cursorPoint. In this expression, view is the View associated with the MVC triad. The View takes the point returned by Sensor cursorPoint and transforms it from screen coordinates to view coordinates. If the middle button is pressed, (Sensor yellowButtonPressed ifTrue:) it executes the second block and calls on the model to execute the method removeFirst.

This example, while it does work, is still buggy. If you hold the left mouse button down, each time the controller gets control, it adds a point then relinquishes it untill the next time it gets polled. Similarly, if you hold the middle mouse button down, you'll delete points until your model is empty, then get a complaint. While not exactly the behavior you might want, experiment with this application. Try to change just how it works. Some avenues to explore are:

- Change the way the points are presented. How would they look if you drew a spline curve through them? Connected them with a line?
- How can you make the controller add or remove only a single element no matter how long you hold down the button? (If you need a hint, try typing yellowButtonPressed in the text pane of the workspace, highlight it, and choose 'explain' from the middle button menu.)
- Create the "philosophically correct" interface. Create a method that lets you select points with the left button. Pop up a menu with the middle button that lets you add points with the left button, remove selected points, or remove the first point or remove the last point. You'll want to look at class ActionMenu to see how to proceed.

Section 8

Operating System Interface

INTRODUCTION

Tektronix Smalltalk provides access to the full capabilities of the underlying operating system. Access includes file system functionality as well as other capabilities of the operating system such as multi-tasking. Applications may use these capabilities to reach outside communication channels, to reach specialized computing power, or simply to execute normal functions of the operating system from within the programming environment. This operating system interface provides a uniform method of accessing many features of the operating system from the programming environment.

Uniform access to operating system capabilities is furnished by the execution of operating system calls. It is extended by abstractions that encompass important operating system facilities.

This discussion assumes that you are familiar with Smalltalk and multi-tasking in the Unix system. Smalltalk has its own threads of control called processes, but at the same time Smalltalk itself is a task (process) in the UTek operating system. For this discussion, an operating system *process* is referred to as a *task.*

This discussion also assumes you are familiar with files, pipes, and signals in a Unix environment; however, as a quick review, you can look at the *Review of OS Terms* provided later.

Overview

This section gives you a basic understanding of how files, pipes, and tasks are treated in Smalltalk. General concepts and orientation are given along with examples. After you have read through this section, go on to the class and method comments and browse the method code in the system.

Review of OS Terms

Here is a quick definition of the terms used in the discussion of the operating system interface classes. A textbook about the Unix operating system will help you review these concepts in detail. A good text is: James Peterson and Abraham Silberschatz. *Operating System Concepts.* (Addison-Wesley, 1985).

kernel	The UTek operating system. It is the software layer between the hardware and the programming interface. The kernel services fall into four categories: tasks, I/O, memory management, and timers.
system call	A request for a kernel service. A system call is the programming interface to the kernel. Tektronix Smalltalk supports a number of UTek system calls.

task	A task is a thread of control in the operating system. Each user can own multiple threads of control, thus, the term multi-tasking. A task may be created, interrupted, waited on, or terminated.
fork	*fork* system call. A *fork* call copies the parent task to create a new task – the child task.
exec	The UTek *exec* system call is used after a *fork* call to turn a child task into a new "program". The *exec* call does this by overlaying the text and data segments of the parent task. But, the child task still has access to the parent's files because the task environment is not changed. So, a *fork* and then an *exec* call creates a new task and runs a program.
signal	The UTek kernel sends a signal to a task in response to events like a a child task termination or program errors. A task deals with system signals by accepting the system default, by ignoring signals, or intercepting the signal with a signal handler.
pipe	A UTek pipe is a uni-directional stream of bytes connecting two tasks. A pipe has two descriptors: one for the read end and one for the write end. Applications must have two pipes for *two* way communication between tasks.
descriptors	A descriptor is an identifier for a stream of data. Pipes and files have descriptors. Each task has a fixed-size descriptor table, which is part of a task's environment. Descriptors are used in read or write calls on pipes and files. There is a limit on the number of descriptors available per task.

THE OS GLOBAL VARIABLE

OS is a global variable that refers to the system call class for your operating system. Currently, Tektronix Smalltalk supports two operating systems: the Uniflex operating system and the UTek operating system.

OS refers to either UTekSystemCall or UniflexSystemCall depending on the operating system you are running. Smalltalk runs on top of either UTek or Uniflex. So, to provide portability of system call functionality between systems, you should use the global variable OS instead of a specific system call class.

THE SYSTEM CALL INTERFACE

Tektronix Smalltalk has added system call functionality to Smalltalk with these classes: AbstractSystemCall, AimSystemCall, UniflexSystemCall, and UTekSystemCall. These classes add abstract operating system functionality by defining operations and capabilities in super classes. Specific functionality is defined in system call classes for specific operating systems.

AbstractSystemCall

AbstractSystemCall is meant to be a model of the basic system call functionality regardless of the underlying operating system. However, the operating system *is* assumed to have multiple directories, files that can be randomly read, written, and truncated, that have some form of status information, and some means of running different programs one at a time. Multi-tasking is not assumed.

Protocol defined for the class AbstractSystemCall determines system call functionality in three categories: directory, file, and subtask operations. The methods may be implemented in AimSystemCall and UTekSystemCall. If possible, applications should base their code on methods defined in AbstractSystemCall. These methods are considered most portable.

AimSystemCall

AimSystemCall is another abstract class intended to implement system call functionality across the 4400 family machines. Through this class you can access 4400 family display operations, operating system environment variables, and command line arguments from the invocation of Smalltalk.

An instance of AimSystemCall contains all of the information normally used to perform a system call from an assembly language interface. This includes:

- The values of machine registers passed to or returned from the system call and any parameter lists.
- The operation ID of the system call to be performed.
- Any error information.

Thus, AimSystemCall includes instance variables named D0In, D0Out, A0In, A0Out, and so forth. These represent 68000 family registers. The instance variable errno returns error information.

UTekSystemCall

UTekSystemCall is a concrete class that implements access to many of the system calls available in the UTek operating system.

Portable Operations

Portable operations are generic or high-level operations that all operating systems at the specified level support in some manner. For example, portable operations specified at AbstractSystemCall are supported by all operating systems; portable operations specified at AIMSystemCall are supported by the UTek and Uniflex operating systems; and portable operations specified at the UTek level are supported by all versions of the UTek operating system. Subclasses may override an implementation of a portable operation.

Always use a portable operation if possible to insure portability across operating systems. (Use of the OS global variable also helps insure portablility.)

For an example of why you should use portable operations whenever possible, take a look at the following method for a common file operation: opening a file for writing. The method openForWrite: is a portable file operation that sends a system dependent message to accomplish its task. Each operating system may have a different way to open a file. The portable operation provides a way to hide these differences. Note that the system dependent method, open:flags:mode:, involves operating system details that you must research and correctly use to accomplish the conceptually simple task of opening a file for writing. However, the correct values for many UTek operating system constants have been determined and are used in portable operations. The constants O_WRONLY and O_CREAT, along with many other operating system constants, are found in the global dictionary OSConstants.

```
openForWrite: fileName
 "Open the file named fileName. Answer a writeOnly fileDescriptor for the file."

 ↑(self
  open: fileName
  flags: ((self constant: 'O_WRONLY')
     bitOr: (self constant: 'O_CREAT'))
  mode: 8r666) value
```

System Dependent Calls

System dependent calls are specific to a particular operating system. They specify the system call and set up the parameters required, but stop short of actually executing the system call. An instantiated instance of UTekSystemCall is produced by system dependent methods. This is a form of instance creation. The messages value, valueWithError:, or invoke are sent to the instantiated instance to execute the specified system call.

Take a look at UTekSystemCall unlink: for an example of a system dependent operation.

```
unlink: pathName
  "Unlink removes the reference to pathName from its directory.
  The file will not go away if there are other links to it or it is open
in any process."

  ↑(self new)
    operation: #unlink
    with: pathName, StringTerminator; systemOperation
```

unlink: creates an instance of UTekSystemCall and sends operation:with: to it. operation:with: specifies the system call *unlink* and sets up the arguments for a system call (here the argument is the result of concatenating pathName with StringTerminator). StringTerminator is a string containing the string termination characters for the underlying operating system – for UTek, a null character. systemOperation informs UTekSystemCall which primitive to execute: a display, a system, an environment, or a signal operation. This example uses systemOperation, which means make a system call.

A sender of unlink: is remove: in AimSystemCall class. System dependent methods are frequently used by portable operations like remove:. Note that the value message is used in

AimSystemCall remove: to actually execute the system call.

Here is an example of a system dependent call that is not needed in Smalltalk and is not used as part of a portable operation call. Look at the following method found under system-files:

```
flock: fileID operation: operationID

"Flock controls advisory locks that cooperating processes may associate
with files. Locks can be applied either exclusively or shared, and may be
set non-blocking. OperationID is a set of bit flags used to determine the
type of lock that will be applied to the open file descriptor fileID."

  ↑(self new)
   operation: #flock
   with: fileID
   with: operationID; systemOperation
```

You might use this method if you were working on a database application in which files need to be protected and shared at the same time.

If you cannot figure out how to use this method, or any other system dependent method, from examination of the code and reading the method comment, then look up the corresponding system call in the *UTek Command Reference* manual. The information in the *UTek Command Reference Volume 2* manual should give you the information you need to use the Smalltalk method since there is a close correspondence between system call specification and system dependent methods.

System Call Parameters

UTek and other operating systems make use of single integers, collections of integers, and sometimes pointers to pass information. Since these data are of various kinds, they are named aggregates for this discussion. The term "binary aggregates" refers to data with no pointers. The term "pointer aggregates" refers to data with pointers.

To support these aggregates, the image contains an abstract class, ExternalData. This class has two subclasses, ExternalBinaryData and ExternalPointerData. Looking at the hierarchy of ExternalData, you will see that most of the C structure classes are subclasses of ExternalBinaryData. These include Stat, Rusage, and Wait. Under ExternalPointerData, you will find PointerArray, StructureArray, and FixedSizeExternalPointerData, the last of which CPointer is an example.

```
ExternalData
    ExternalBinaryData
        . . .
        Stat
        Tchars
        Timeval
        Timezone
        Utsname
        Wait
    ExternalPointerData
        FixedSizeExternalPointerData
            Iovec
            Msghdr
        PointerArray
        StructureArray
```

Naming Conventions

A class is defined for each aggregate required by UTek system calls. The name of the class is created by capitalizing the first letter of the corresponding aggregate name. Underscores are translated into Smalltalk style capitalization conventions. Thus, the correspondence of C structures and their Smalltalk counterparts is readily apparent. Protocol is provided to create instances of the classes representing aggregates and to access field elements by name. Selectors for accessing structure fields correspond to the C field name with underscores translated. Here is an example:

The C structure declaration:

```
struct sockaddr_in {
  short sin_family;
  u_short        sin_port;
  struct in_addr sin_addr;
  char           sin_zero[8];
};
```

The Smalltalk definition of the class SockaddrIn:

```
ExternalBinaryData variableByteSubclass: #SockaddrIn
  instanceVariableNames:''
  classVariableNames: 'AddrDataIndex FamilyDataIndex PortDataIndex
ZerosIndex ZerosLength'
  poolDictionaries:''
  category: 'OS-Parameters'!
SockaddrIn comment:
```

The correspondence between Smalltalk structure names and C structure names should help you use the *UTek Command Reference* manuals as documentation for system dependent methods.

Here is an example of a system call that needs a structure passed to it. Look for the *wait* system call either online in manual page format or in the UTek reference documentation under system calls. Under the SYNOPSIS heading, you find that the *wait* system call requires a *union wait* pointed to by *status*. The class Wait in category OS-Parameters forms a Smalltalk equivalent to this union. See the comment in the browser under Wait for more information.

Also, look at the method wait in UTekSystemCall class to see how the class Wait is used.

Refer to the *Tektronix Smalltalk Reference* manual for information on specific ExternalData classes.

Signals

Signals, or interrupts, may be sent to tasks in many operating systems. Both the UTek and Uniflex operating system support sending signals. The portable operation:

```
OS sendInterrupt: interruptID to: taskID
```

may be used to send a signal to the task specified by taskID. The first argument interruptID is a specially constructed numeric identifier. Signals may also be sent back to Smalltalk. A restricted signal handler may be constructed in Smalltalk with the message:

```
OS setInterrupt: interruptID to: aSemaphore
```

The semaphore is signaled upon interrupt receipt. Thus, a Smalltalk process may connect to the signal and suspend itself until the signal is received via the sequence:

```
OS setInterrupt: interruptID to: aSemaphore.
aSemaphore wait
```

If the Smalltalk process wishes to receive the signal more than once, it may need to reconnect the signal and semaphore. You can also ignore and set signals to default actions with the methods:

```
OS ignoreInterrupt: interruptID

OS defaultInterrupt: interruptID
```

Interrupt Identifiers

An interrupt, in UTek, is composed of a signal and a parameter. The UTek Smalltalk interpreter, to support the semaphore/interrupt connection, maps a signal and its parameter into a numeric interrupt identifier. These interrupt ID's are used by the interpreter primitive #133 to construct a signal handler which supports the semaphore/interrupt connection. The methods setInterrupt:to:, ignoreInterrupt:, and defaultInterrupt: require these special interrupt identifiers.

The following table lists interrupt identifiers. The interrupt ID's are the numbers on the lefthand side of the following table. The UTek signal number and its parameter are in the second column.

Table 8-1
Interrupt Identifiers

Interrupt ID	UTek Name	UTek#+parm
0	N/A	
1	SIGHUP	1
2	SIGINT	2
3	SIGQUIT	3
4	<reserved>	
5	<reserved>	
6	<reserved>	
7	<reserved>	
8	<reserved	
9	SIGKILL	9
10	SIGBUS	10
11	SIGSEGV	11
12	SIGSYS	12
13	SIGPIPE	13
14	SIGALRM	14
15	SIGTERM	15
16	SIGURG	16
17	SIGSTOP	17
18	SIGSTP	18
19	SIGCONT	19
20	SIGCHLD	20
21	SIGTTIN	21
22	SIGTTOU	22
23	SIGIO	23
24	SIGXCPU	24
25	SIGXFSZ	25
26	SIGVTALRM	26
27	SIGPROF	27
28	SIGUSR1	28
29	SIGUSR2	29
30	SIGWINCH	30
31	SIGPWR	31
32	ILL TRAP	4+0
33	PRIV TRAP	4+1
34	ALINE TRAP	4+2
35	FLINE TRAP	4+3
36	FORMAT TRAP	4+4
37	BKPT TRAP	5+0
38	TRC TRAP	5+1
39	CHK TRAP	6+18
40	TRAPV TRAP	6+1C
41	TRAP0	6+80
42	TRAP1	6+84

Table 8-1 (cont.)
Interrupt Identifiers

Interrupt ID	UTek Name	UTek#+parm
43	TRAP2	6+88
44	TRAP3	6+8C
45	TRAP4	6+90
46	TRAP5	6+94
47	TRAP6	6+98
48	TRAP7	6+9C
49	TRAP13	6+B4
50	TRAP14	6+B8
51	TRAP15	6+BC
54	INTDIV TRAP	8+1
55	FLTOVFL TRAP	8+2
56	FLTDIV TRAP	8+3
57	FLTUNDFL TRAP	8+4
58	INEXACT TRAP	8+5
59	BRCOND TRAP	8+6
60	NAN TRAP	8+7
61	OPERAND TRAP	8+8
62	<reserved>	
63	<reserved>	

COMMUNICATION CHANNELS

Smalltalk passes streams of bytes back and forth to the operating system mainly through two communication channels: files and pipes. The Smalltalk classes that represent these operating system structures behave in essentially the same way as their UTek counterparts. There are some differences, though, in how Smalltalk files work. The Tektronix Smalltalk abstraction of UTek (or Uniflex) pipes emulates their behavior exactly.

Files

Files can be opened or closed, read from or written to, have file position altered, etc. They are different from pipes in that data sent to files is not lost when files are properly opened and closed. Pipes, existing only in memory, can lose information if they are not properly set up, opened and closed, or flushed.

FileStream

An instance of FileStream represents a file. Ordinarily, the file is not opened until some I/O is performed with the corresponding instance of FileStream. This is the case with the instance creation message fileNamed:. In the following sequence of events, the next message causes the file to be opened.

```
aFileStream ← FileStream fileNamed: 'readTest'.
aFileStream next
```

The instance creation messages newFileNamed: and oldFileNamed: cause immediate opening of the file to ensure a new or old file, respectively. The fileNamed: method always executes quietly. The newFileNamed: message assumes the corresponding file does not exist and notifies the user if the file does exist. Proceeding renames the existing file. The oldFileNamed: message assumes that the corresponding file does exist and notifies the user if it does not. Proceeding creates the file. These instance creation messages allow applications some control over the existing state of the file system. For example, in some cases it would be wrong to overwrite an existing file. In this case, the application should use the newFileNamed: message to ensure no existing files are overwritten without notification.

File descriptors arre a limited resource in most operating systems. To accommodate this limitation, instances of FileStream can have their corresponding files closed and reopened without loss of data or state. Without users' knowledge, when the Smalltalk task runs out of file descriptors, it automatically closes the necessary number of files. These files are automatically reopened the next time they are referenced.

FileDirectory

Instances of FileDirectory are a special kind of FileStream, representing directories in the file system. Instances can query the file system to find out what files its directories contain. Instance protocol includes enumberating, copying, and appending methods. Pattern matching protocol is also available. For example:

```
files ← someDirectory filesMatching: 'A*'
```

The returned value contains a collection of all files in the directory beginning with the character "A". The character "*" (asterisk) matches multiple character. The character "#" (pound sign) matches any single character.

The global variable Disk represents the Smalltalk home directory. All "fileouts" go to Disk. Relative paths are defined with respect to Disk.

FileDirectory has instance methods with the same functionality as FileStream class methods. These methods create new instances of FileSteam. See the table below.

Table 8-2
Parallel FileDirectory and Filestream Protocol

FileDirectory	FileStream
oldFile:	oldFileNamed:
newFile:	newFileNamed:
file:	fileNamed:

Using FileStreams and FileDirectories

To open a previously existing file for reading or writing, send the message oldFile: with the file name as an argument to an instance of a FileDirectory such as:

```
f ← Disk oldFile: 'timingData'
```

Executing this expression insures that the variable f is an instance of a FileStream on an existing file. A notifier appears if the file does not exist.

To open a new file, send the message newFile: to an instance of a FileDirectory as in:

```
f ← Disk newFile: 'testCases'.
```

This guarantees that f is a FileStream on a new file named *testCases*. If a file in the Disk directory previously exists with that name, a confirmer tells you that the file already exists. If you proceed, the existing *testCases* file is renamed as *testCases~* and a new *testCases* file is created.

The file: message delays the opening of a file until a data transfer takes place.

```
f ← Disk file: 'testResults'
```

If such an instance of a FileStream is sent the nextPut: message, and a file with that name already exists, the file is automatically backed up with a ~ (tilde) extension and the data is written to the new file. If the file *testResults* does not exist and a read operation is attempted, an error is produced. If the file *testResults* does not exist and a write operation is attempted, the file is created, opened for reading and writing.

One can query the existence of such a file before doing input or output. For example:

```
f ← Disk file: 'testResults'.
f exists
```

answers true if the yet-to-be opened file already exists.

Sending the message readWrite to a FileStream created with the file: message insures that no backup is made. This allows updates in place.

For special devices like the sound device, */dev/sound*, which can only be opened for writing, use the oldWriteOnlyFile: message.

```
f ← Disk oldWriteOnlyFile: '/dev/sound'.
f binary.
f nextPut: soundDevByteArrayBuffer.
f close.
```

The oldWriteOnlyFile: message says that this file is intended to be a previously existing file. (Thus, do not employ the back-up mechanism, which is to be opened strictly in a writeOnly mode.)

The Smalltalk interface to the file system is sophisticated enough to manage the opening and closing of files fairly transparently. An attempt to read a closed file causes the file to be opened. If an attempt is made to open a file that is referenced by another open FileStream, an attempt is made to close the already open file. Its position is retained so that later data transfers can resume unaffected by the closure.

It is still a good idea to close files explicitly. Closing forces all buffers to be flushed to the physical file, thus protecting from unexpected data loss in the event of a crash.

Pipes

A pipe is a UTek operating system facility for communicating between two processes (or tasks). Pipes are one-way devices. A byte stream passes through it in one direction only. One process writes to one end of a pipe, and the other process reads from the other end of the pipe. Pipes have one write file descriptor designating the write end, and one read file descriptor designating the read end of the pipe. (That is, there are two descriptors total for each pipe.) Pipes exist entirely in memory, so some care must be taken to be sure all data has passed through the pipe before it is closed, since closing a pipe results in loss of any data in the pipe.

Pipe

Instances of class Pipe are created by executing the expression Pipe new. In Smalltalk, Pipes are used to communicate between Smalltalk itself which exists as a UTek process and some other process in the operating system. Pipes can be opened and closed only once. So, you need to create a new pipe each time you expect to open a pipe. See the discussion of subtasks and pipes later.

PipeStream

The act of creating a Pipe is not sufficient to communicate with an operating system task. The Smalltalk side of the pipe needs a kind of PipeStream created and connected to it. PipeReadStreams or PipeWriteStreams act as the source or destination of data being sent through the Pipe or received from the Pipe. PipeStreams inherit protocol from Stream, PositionableStream, WriteStream, etc. PipeStream itself is an abstract class, so, you will actually create instances of PipeReadStream or PipeWriteStream in code.

PipeReadStream buffers its data, whereas PipeWriteStream does not. Pipe streams know about file descriptors so they can connect to Pipes. Smalltalk makes a distinction between text and binary data, so PipeStreams know about data mode also. All instances of PipeStreams can access the class variable OpenPipeStreams.

A check in the browser reveals that AimSystemCall, OSFilter, and the SystemDictionary all use PipeReadStream. PipeWriteStream is used in the class OSFilter. Since Pipes and pipe streams are used almost entirely in conjunction with Subtasks in the system, see later under *Multi-tasking Concepts* for illustrations of how Pipes and PipeStreams are used.

MULTI-TASKING CONCEPTS

Smalltalk runs on top of an operating system, such as UTek, as a task. Since the UTek operating system supports multi-tasking, Smalltalk takes advantage of this by supporting the creation, execution, communication, and termination of operating system tasks, too. Tektronix Smalltalk does this through a combination of resources in the classes Subtask, Pipe, and PipeStream and its subclasses.

By using these classes, you can run operating system utilities as well as communicate with applications and programs written in other languages. Tasks can be executed without leaving the

Smalltalk environment. Interfaces to operating system signals, program parameters, environment variables, and subtask priorities are also supported.

The class OSFilter makes use of the Smalltalk multi-tasking capability. OSFilter is examined later in this section.

Subtasks

The class Subtask gives Smalltalk the ability to spawn and manage multiple operating system tasks. Subtasks may be created, transformed into useful programs, and terminated. Multiple tasks may be created and managed from within Smalltalk, so you can check on task IDs, task priorities, and other task status.

Subtasking Examples

Here are some simple, straightforword examples that demonstrate how to use executeUtility:withArguments:. These are followed by discussion of how subtasks get created, how they are transformed into user programs, how they may be suspended, and how they are managed. Each argument is an element in the OrderedCollection. This example returns a list of the files in the */usr/lib* directory.

```
flags ← '-sa'.
OS executeUtility: '/bin/ls'
  withArguments: (OrderedCollection with: flags with: '/usr/lib').
```

Shells have capabilities that many utilities do not individually implement. The next example executes a shell with a 'c' option. The 'c' option tells the shell to read the rest of the arguments as a command to itself. The effect is a directory listing with the shell providing wildcard expansion and search path capability.

```
pattern ← '/usr/lib/smalltalk/file*'.
nameList ← OS
    executeUtility: '/bin/csh'
    withArguments: (OrderedCollection
        with: '-c'
        with: 'ls -s ' , pattern)
```

Besides taking advantage of the shell's wildcard expansions, you can use aliases (also implemented by the shell):

```
OS executeUtility: '/bin/csh'
   withArguments: (OrderedCollection with: '-c' with: 'll')
```

where ll is an alias for ls -a.

To execute a program with no arguments, substitute an empty OrderedCollection for the argument.

```
OS executeUtility: '/bin/uptime'
  withArguments: OrderedCollection new.
```

Subtask Creation and Program Execution

An instance of Subtask represents a spawned operating system task that loads and executes a binary file, which may be a UTek command, a user-written application, or other program. There are five parts to the process of creating a subtask and running the program it turns into:

1. Set up and creation of a Subtask instance – a child task of the (original) parent Smalltalk task.
2. Set up of any operating system or Smalltalk structures (having to do, usually, with inter-task communication and signal handling) needed by the Subtask's (child subtask's) binary file (program).
3. Transformation of the child subtask into the binary file (program) and execution of that program.
4. Optional suspension of the Smalltalk task while the child task's program executes.
5. Return to the Smalltalk task after successful or unsuccessful execution of the child task.

See *Figure 8-1, Subtask Creation and Program Execution* as you go through the text.

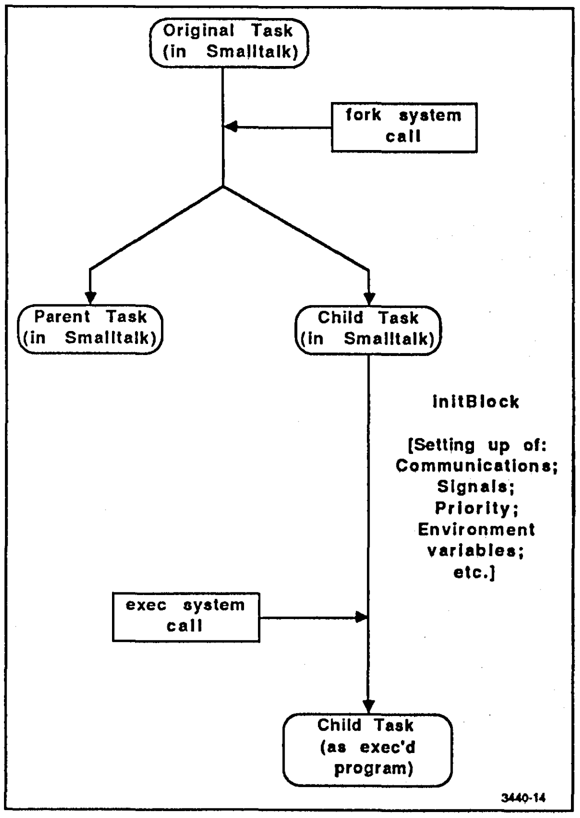

Original Task (in Smalltalk)
fork system call
Parent Task (in Smalltalk)
Child Task (in Smalltalk)
initBlock
[Setting up of: Communications; Signals; Priority; Environment variables; etc.]
exec system call
Child Task (as exec'd program)
3440-14

Figure 8-1. Subtask Creation and Program Execution.

Here is a simple, *conceptual* example of this process.

```
task ← Subtask fork: '/bin/simpleUtility' then: [].
task start.
task wait.
```

The first line creates a new instance of Subtask and adds the operating system command *simpleUtility* to the Subtask's OrderedCollection instance variable arguments. *simpleUtility* might be something like *ps* or *ls* with no arguments.

The second line actually creates the operating system task by spawning it as a child of the Smalltalk parent task. Any communication or signal processing is taken care of in the initBlock, which here is empty, and the child task is transformed into *simpleUtility*, which begins execution.

The third line requests that the Smalltalk process suspend execution until the completion of the child subtask *simpleUtility*.

Smalltalk monitors and manages subtasks through a subtask management system, which is not apparent in this simple example. See later in this section for more information about Subtask management.

This sequence of messages to Subtask should be followed for most applications of Subtask. There are, however, two basic types of suspension of the Smalltalk parent task.

- You can suspend only a single Smalltalk process with the wait method.
- You can suspend the entire Smalltalk task with the waitWithSmalltalkSuspended method.

wait is the usual choice, but if your subtask is non-interactive and requires a lot of computer resources, you may want to use waitWithSmalltalkSuspended.

Environment Variables

The Smalltalk-80 system's interface to subtasks also supports environment variables. (See the *4405/4406 Option 15 UTek Exceptions and Extensions* manual for more details.) In general, when a program is invoked, the operating system passes arguments and environment variables to the program. Standard environment variables include HOME – a home directory specification and PATH – a search path specification. Environments are a way to pass information by name. This can be viewed as setting a context for execution. Instances of subtask are created with a default environment, the environment with which Smalltalk was invoked. The method Subtask class currentEnvironment answers the default environment. The environment variables are kept in dictionary format for easy modification. The method Subtask environment: assigns an environment to the Subtask instance. The environment is passed to the executed program. Here is an example of use of a modified environment.

```
execProgram: aCommand
  | task env |
  task ← Subtask
        fork: aCommand
        then: [].
  env ← Subtask copyEnvironment.
  env at: #HOME put: '/smalltalk'.
  task environment: env.
  task start isNil
    ifTrue:
      [self error: 'Cannot execute ', aCommand].
  task waitOn.
  task abnormalTermination ifTrue: [self error: 'Error from ', aCommand].
  task release.
```

Interrupting Subtasks

Interrupt messages send a signal to the spawned task. Some of these messages cause termination of the task. *Others merely try to communicate a termination signal to the task.* The spawned task may ignore this request. Here are two important *interrupt* messages:

- terminateUnconditionally – This terminates the spawned task, and it may not be ignored. However, only tasks owned by the user may be terminated unless the user is the super user. (See UTek system administration documentation for what a super user is.)
- interrupt: – This sends the specified interrupt to the spawned task. The default action of many signals is task termination, so be sure the spawned task is expecting the signal.

Signals

Interrupts can be intercepted, ignored, or set to a default action by using protocol in system call classes. Usually, the default action upon receipt of an interrupt is task termination. Sometimes it is desirable for a task to intercept, or modify, its reaction to an interrupt. Spawned subtasks can modify their reaction to an interrupt.

Protocol to specify the action upon receipt of a signal can be added to the block which is an argument to the Subtask instance creation methods. Here the method fork:withArgs:then: is passed a block which modifies some of these reactions. Code in this block is executed by the child task only. The ScreenController forkOSShell method modifies the action of several interrupts with the method setInterrupt:to:. forkOSShell modifies the action upon interrupt in both the parent and child task. Here is a simplified and stripped down copy of that method.

```
forkShell

"Set up the display and signal environment for terminal emulation,
and turn it over to a forked shell Subtask.  Block on the Subtask until it
terminates, then restore the display and signal environment for
Smalltalk."

  | aDisplayReport sigDict task command |
  aDisplayReport ← Display getDisplayReport.
  sigDict ← Dictionary
        with: SIGHUP -> nil
        with: SIGINT -> nil
        with: SIGQUIT -> nil
        with: SIGTERM -> nil.
  command ← self originalEnvironment at: #SHELL.
  command class ~= String ifTrue: [command ← '/bin/sh'].
  FileStream releaseExternalReferences.
  sigDict associationsDo: [:sig | sig value: (self ignoreInterrupt: sig key)].
  task ← Subtask
        fork: command
        withArguments: (OrderedCollection with: '-is')
        then:
          [sigDict keysDo: [:sig | self defaultInterrupt: sig].
          self setDisplayUTek].
  task start isNil
    ifTrue:
      [sigDict associationsDo: [:sig | self setInterrupt: sig key to: sig value].
      self restoreSmalltalkWith: aDisplayReport.
     ↑ self error: 'Cannot execute ', command printString].
  task waitWithSmalltalkSuspended.
  task release.
  sigDict associationsDo: [:sig | self setInterrupt: sig key to: sig value].
  self restoreSmalltalkWith: aDisplayReport
```

Interrupt action is modified in both the parent and child tasks by using the methods:

- setInterrupt:to:
- defaultInterrupt:
- ignoreInterrupt:

which return the previous action for that interrupt. First, the parent interrupt actions are saved in a temporary variable, sigDict, while setting interrupt actions to ignore. In the subtask, these same interrupts are reset to the default action. After the subtask has completed, the interrupts in the parent process are set back to their original values. In this method, the parent task waits for the child task to terminate by using the method waitWithSmalltalkSuspended. This method actually shuts down the Smalltalk parent process so it receives no time slice from the operating system scheduler. This strategy of waiting makes the subtask more efficient because the parent process cannot steal any processing power. However, Smalltalk cannot run until the child task has terminated. The message waitWithSmalltalkSuspended is not appropriate for any subtask that depends on the Smalltalk user interface.

Waiting for Subtasks

There are two types of waiting, and neither one is mandatory.

- One kind suspends only one process – the Smalltalk process controlling the subtask. The process is suspended until the spawned task terminates. Suspension is initiated by the message wait to an instance of Subtask and implemented with a Smalltalk semaphore.
- The other kind suspends the entire Smalltalk task including all Smalltalk processes. This is used when a spawned task needs all the system resources *and* when there is no interaction between Smalltalk and the spawned task. This kind of waiting is initiated by the message waitWithSmalltalkSuspended.

Restarting After a Snapshot

If you have created a Subtask and a start has spawned the child subtask, the status of the child subtask may be #running, #waitedOn, #terminationSignaled, etc. Note that this is *not* the status of the executing binary file (program) that the child task is transformed into but is the status of the Smalltalk subtask object. (See *Subtask in Detail* later.) Regardless of what state the child subtask is in, however, the status goes to #nonExistent if you do a snapshot. Thus, when you reload the image after a snapshot all child subtasks will have status nonExistent. This means that applications can use this information to restart Subtasks after a snapshot.

The following methods found in the instance protocol category, testing, for Subtask may be used to determine what state child tasks are in. These methods answer true if:

- abnormalTermination – Some sort of error in execution has occurred and the subtask has terminated abnormally.
- isActive – The subtask is operating normally.
- isNonExistent – There has been a snapshot since the subtask was created.
- isTerminated – The subtask has terminated whether abnormally or not.
- notTerminated – The subtask has not yet terminated.

Subtask in Detail

This subsection explains the messages that are sent when you create, transform, and execute a new process from within Smalltalk.

By doing a hierarchy menu command in the System Browser, you can see that Subtask inherits only from Object. You will also note from the System Browser that Subtask has a fair number of instance and class variables. The more important instance variables are defined here. You can see all of the variable definitions by using the System Browser or by looking them up in the *Tektronix Smalltalk Reference* manual under Subtask.

status — A Symbol indicating the state of a task. The more important values are:

#running	A task has control of the CPU and is accomplishing its job.
#waitedOn	A task is waited for by the controlling Smalltalk process.

	#nonexistent	A task goes to this state when you do a snapshot.
program	A String containing the path of the program to be executed. For example, */bin/ls*.	
arguments	An OrderedCollection of Strings, each of whose values is an argument to the program. For example, if the progam is *ls*, arguments might include *-F* or *-a*.	
environment	A Dictionary of operating system environment variables, keyed by environment variable. Dictionary values are the values of the environment variables.	
initBlock	A Block to be executed between the *fork* call and the *exec* call. Usually this block concerns signals and communication. See later under Subtask creation for more information about this.	

The following discussion takes a real example from the system that accomplishes something very simple. It creates a UTek shell, from which you can return to Smalltalk when you are through. Look in the System Browser under UTekSystemCall class. Here you find a selector, forkShell, in the protocol category portable subtask operations.

If you want to see how the method works, open a workspace, type in OS forkShell, and do a do it. You see that from Smalltalk you have started up a UTek shell. Execute some UTek commands to see that you have indeed forked a shell process, and then return to Smalltalk with an *exit* command or <Control-d>.

Take a look now at the forkShell method.

There are four major parts to this method:

1. Signal setup code from the beginning to sigDict associationsDo:....
2. A Subtask setup part, task ← Subtask...
3. A Subtask execution part from task start isNil... to task release
4. Signal and display restore part at the end.

This discussion focusses on parts 2 and 3.

```
forkShell

  "Set up the display and signal environment for terminal emulation,
  and turn it over to a forked shell Subtask. Block on the Subtask
  until it terminates, then restore the display and signal environment
  for Smalltalk."

  | aDisplayReport sigDict task command |
  aDisplayReport ← Display getDisplayReport.
  sigDict ← Dictionary
        with: SIGHUP -> nil
        with: SIGINT -> nil
        with: SIGQUIT -> nil
        with: SIGTERM -> nil.
  command ← self originalEnvironment at: #SHELL.
  command class ~= String ifTrue: [command ← '/bin/sh'].
  FileStream releaseExternalReferences.
  sigDict associationsDo: [:sig | sig value: (self ignoreInterrupt: sig key)].
  task ← Subtask
        fork: command
        withArguments: (OrderedCollection with: '-is')
        then:
          [sigDict keysDo: [:sig | self defaultInterrupt: sig].
          self setDisplayUTek].
  task start isNil
    ifTrue:
      [sigDict associationsDo: [:sig | self setInterrupt: sig key to: sig value].
      self restoreSmalltalkWith: aDisplayReport.
      self error: 'Cannot execute ', command printString].
  task waitWithSmalltalkSuspended.
  task release.
  sigDict associationsDo: [:sig | self setInterrupt: sig key to: sig value].
  self restoreSmalltalkWith: aDisplayReport
```

Part 2 Subtask Setup. Here is part two of the method given above.

```
task ← Subtask
      fork: command
      withArguments: (OrderedCollection with: '-is')
      then:
        [sigDict keysDo: [:sig | self defaultInterrupt: sig].
        self setDisplayUTek].
```

The method fork:withArguments:then: is an instance creation method to Subtask. Here the default binary file this subtask will execute is */bin/sh*, which is the value of the argument command. The UTek arguments to */bin/sh* are readily apparent as *-is* from the expression OrderedCollection with: '-is'. The argument to fork:withArguments:then: is the initBlock. The initBlock is the place where you put code to be executed *before* the specified binary program begins executing. The initBlock usually contains communication setup and signal processing or handling code. If you expect to receive data back from a UTek command, for example, you would set up Pipes and pipe descriptors here. In this example, the initBlock handles possible interrupt signals and sets up the display system to act as an appropriate command line interface for UTek.

Part 3 Subtask Execution. Here is part three of the method given above.

```
task start isNil
    ifTrue:
      [sigDict associationsDo: [:sig | self setInterrupt: sig key to: sig value].
      self restoreSmalltalkWith: aDisplayReport.
      self error: 'Cannot execute ', command printString].
  task waitWithSmalltalkSuspended.
  task release.
```

Sending the message start to task causes a shell to be spawned and control turned over to it. The result of task start is checked for nil. A return value of nil means a Subtask cannot be spawned or a program executed. If not, the block after the ifTrue: is executed. In this block, the setup code in part 1 is essentially undone and you are shown the error message "Cannot execute" concatenated with the program name.

The method start actually performs the Subtask program set up and request. Look at the code for start now.

```
start

"Start the receiver by spawning a child, executing code to set up the
child task (mainly communication and signal processing), and executing the
program. If the execute fails terminate the child task. The child task
will inherit the priority of the smalltalk task."

  | execer childBlock id |
  execer ← OS execute: program withArguments: arguments withEnvironment: environment.

  childBlock ← [initBlock value.
    priority notNil ifTrue: [OS setTaskPriority: priority].
    FileStream closeExternalReferences].

  id ← OS startSubtask: execer withBlock: childBlock.
  self taskID: id.
  id isNil ifTrue: [↑nil].
  self class addSubtask: self
```

OS is a global variable standing for the system call class appropriate for the operating system underlying Smalltalk. The arguments to execute:... in the first line of code, program, arguments, and environment, are instance variables of Subtask. In this example, program is the String '/bin/csh'; arguments is the OrderedCollection ('/bin/csh' '-is'); and environment is the Dictionary (TERM->'peg-norm' HOME->'/usr/keithr' PATH->':/bin:/usr/bin' USER->'keithr' SHELL->'/bin/csh').

Another instance variable of Subtask, initBlock, makes up part of the temporary variable childBlock. At the time the childBlock is evaluated, initBlock is evaluated. Here initBlock is composed of:

```
[sigDict keysDo: [:sig | UTekSystemCall defaultInterrupt: sig].
          UTekSystemCall setDisplayUTek]
```

This is found in the forkShell method earlier. Some other jobs preparatory to executing the Subtask are done in the childBlock having to do with priorities and FileStreams.

In the next line of code execer and childBlock are used as arguments to another method implemented in UTekSystemCall. Here is the code for this method:

```
startSubtask: execCall withBlock: childBlock

"Fork a copy of Smalltalk.  In the child copy, execute childBlock and
invoke execCall, which must be an instantiated 'exec' system call.  If
execCall returns, there is an error: terminate the child task.  Meanwhile,
the parent task returns the child task ID."

  | syscall pid |
  syscall ← OS fork.  "copy of current Smalltalk process"
  pid ← syscall value.
  syscall D1Out = 1 ifTrue:            "This must be the child."
    [childBlock value.
    execCall invoke ifFalse:  "perform the actual system call; invoke found in AimSystemCall"
      [(OS exit: self abnormalTerminationCode) value]].
  ↑pid
```

With this method, you are at the point where a subtask is finally created. fork sets up the *fork* system call request to create a copy of the Smalltalk operating system task itself. value in the next line sends the message invoke, which actually performs the system call.

If the value in the D1 register indicates that the task is the newly created task, the expressions in the childBlock are evaluated with the message value. Recall that childBlock handles some signal and display related set up.

execCall is an *exec* system call request for an operating system shell to be run (*/bin/sh*). The Smalltalk child task is transformed into an operating system shell in this example. There is no return from a successful *exec* call. If the running task is the parent task, the value returned in D1 is the ID of the child task. The child task's ID is returned so that the Subtask management can record it. Appropriate error handling code completes this method.

Subtasks and Pipes

Pipes are created and used during the creation and execution of Subtasks to send output or receive input. Pipe streams – either PipeReadStreams or PipeWriteStreams – are used to read from or write to the appropriate ends of Pipes. File descriptor assignments determine which end is the "write" end and which end is the "read" end of a Pipe.

Follow this general sequence of operations to use Pipes with Subtasks.

1. Create the Pipe (or Pipes) you need.
2. Create the Subtask with the program you want and any arguments the program needs. In the initBlock, be sure to assign file descriptors to the Pipe.
3. Start up the subtask and close the Pipe if there are any errors in start up.
4. If you expect to collect input from the Pipe, create the pipe stream (or pipe streams) you need and open them on your Pipe(s).

5. Close the PipeStream(s) explicitly.

See *Figure 8-2, Subtask Communication and Pipes,* for a general picture of the relationship between the parent Smalltalk task and the executing child task.

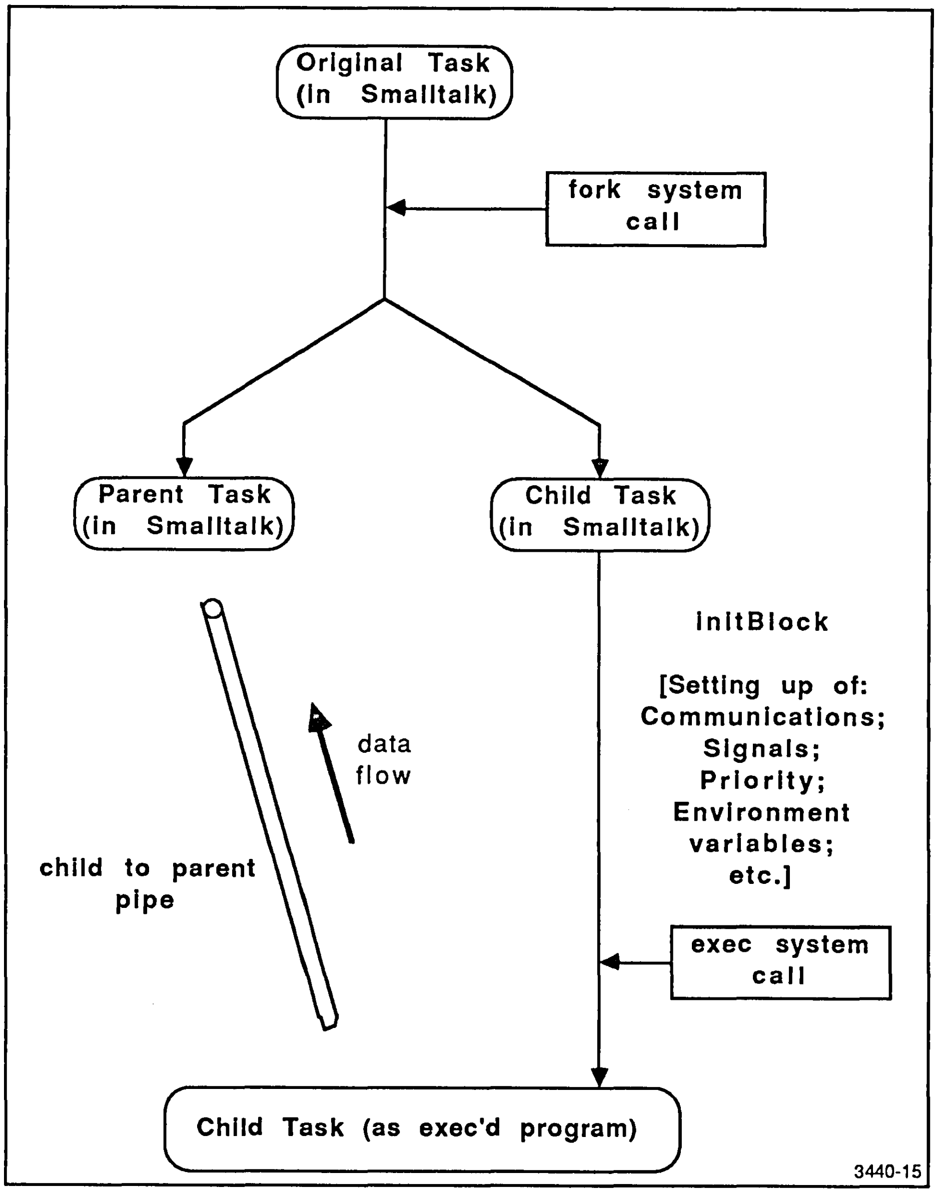

Original Task (In Smalltalk)
fork system call
Parent Task (In Smalltalk)
Child Task (In Smalltalk)
InitBlock
[Setting up of: Communications; Signals; Priority; Environment variables; etc.]
data flow
child to parent pipe
exec system call
Child Task (as exec'd program)
3440-15

Figure 8-2. Subtask Communication and Pipes.

A Schematic Subtask Example with Pipe

The following example method is *not* found in the Smalltalk system, but it *does* show you in a general way how Pipes are handled in conjunction with Subtasks.

```
executeUtility: aCommand

    | pipe task inputSide resultOfProgram |
    pipe ← OS newPipe.
    task ← Subtask
          fork: aCommand
          then:
            [pipe mapWriteTo: 1.
            pipe mapWriteTo: 2.
            pipe closeWrite; closeRead].
    task start.
    pipe closeWrite.
    inputSide ← PipeReadStream openOn: pipe.
    resultOfProgram ← inputSide contentsOfEntireFile.
    task wait.
    inputSide close.
    task release.
    ↑ resultOfProgram
```

In this method, executeUtility:, a Pipe is created to establish one-way communication with the Subtask. (Two Pipes are required for two-way communication.) The code in the block is executed by the Subtask after the *fork* system call and before the *exec* system call. (See the heading *Subtask in Detail* earlier for information about *fork* and *exec* calls.) All the rest of the code in executeUtility: is executed by the parent task.

Pipe connections in the child task are established in the block. In this case, the child task's standard output (file descriptor 1) and standard error (file descriptor 2) are redirected to the Pipe through the use of the mapWriteTo: method. When the child task writes to standard output or standard error, this mapping causes the write operations to be directed to the write side of the Pipe. Since the original write descriptor of the Pipe will not be used because of redirection, it is a good idea to close the write end with the message closeWrite. In addition to closing redirected ends of the Pipe in the child task, unused ends of the Pipe should be closed in both the parent and child tasks. In this case, the Pipe read end is unused in the child task, and the Pipe write end is unused in the parent task.

The net effect of all this closing and mapping is that the child task (whose code is executed in the block) closes the read side of the Pipe because it is unused and closes the write side of the Pipe because it has mapped the write side to standard output and standard error. The parent task closes its unused end of the Pipe, which is the write side. The parent task also creates a Smalltalk object for reading from the Pipe, an instance of PipeReadStream called inputSide. inputSide inherits protocol from PipeStream and, consequently, ExternalStream. Although other methods may be used to read from the Pipe, here, the method contentsOfEntireFile is used to read all the data from the Pipe, and the Pipe is closed after use.

A Real Subtask Example With Pipes

The two lines of code in the following example allow you to receive file information from a UTek *ls* command. And, a further examination of executeUtility:withArguments: reveals a typical use of the Subtask methods, similar to the schematic example earlier.

Open a workspace and type in the following two expressions. Make certain that a space is appended to the last string "/bin/ls -s ". The concatenation message "," – a comma – produces this string, */bin/ls -s /usr/lib/smalltalk/fileIn/*.st*, as the last element in the OrderedCollection .

```
fileInformation ← OS executeUtility: '/bin/ls'
  withArguments: (OrderedCollection with: '-s' with: '/usr/lib/smalltalk/fileIn' )
```

Execute the entire expression and then do a fileInformation inspect. You should see a list of all the files in the fileIn directory. Thus, you executed the *ls* command, and Smalltalk received its output as a String. To see how this is done, look at the code in the system for executeUtility:withArguments:.

```
executeUtility: aCommand withArguments: anOrderedCollection

"Execute a binary program and return the entire results generated by the
program as a string. No mechanism for input to the program is provided.
Create an error if the program cannot be executed or if the program
terminates abnormally."

  | pipe task inputSide resultOfProgram |
  pipe ← OS newPipe.
  task ← Subtask
        fork: aCommand
        withArguments: anOrderedCollection
        then:
          [pipe mapWriteTo: 1.
          pipe mapWriteTo: 2.
          pipe closeWrite.
          pipe closeRead].
  task start isNil
    ifTrue:
      [pipe closeWrite.
      pipe closeRead.
      self error: 'Cannot execute ', aCommand].
  pipe closeWrite.
  Cursor execute
    showWhile:
      [inputSide ← PipeReadStream openOn: pipe.
      resultOfProgram ← inputSide contentsOfEntireFile].
  task wait.
  inputSide close.
  task abnormalTermination ifTrue: [self error: 'Error from system
    utility: ', (resultOfProgram copyUpTo: Character cr)].
  task release.
  ↑ resultOfProgram
```

The first line of executeUtility:withArguments: creates a pipe. If you know that you want the operating system program to return data to the Smalltalk system, then you need to create a pipe and set up its descriptors. You also need to close the pipe ends properly just as you would files when you are through with them.

The next line creates a child task which is the ls command with its desired options and file. aCommand here is the String '/bin/ls' and anOrderedCollection contains the Strings: '-s' and '/usr/lib/smalltalk/fileIn'. Note especially that the block argument to the fork:withArguments:then: message sets up the pipe descriptors. 1 is standard output and 2 is standard error. Unused pipe descriptors are closed at this time. The block is used to set up the child task for appropriate communication with the Smalltalk task. This may involve simple signal handling, manipulation of pipes, or manipulation of environment variables.

The block needs to be handled with some care. The Smalltalk system does not read the keyboard or mouse during execution of the initBlock so be sure to avoid methods that do this. Errors produced at this point cannot be debugged.

The task start... expression creates and transforms the child subtask into a shell process that then runs the *ls* command.
This returns the file information output to the pipe. The pipe is then closed and a PipeReadStream is created to eventually transform the information in the pipe into a String, resultOfProgram, which is returned by executeUtility:withArguments:. Note that you need a PipeReadStream to collect data in the pipe.

Subtask Management

A subtask management system is built into the class Subtask. By means of this system, a spawned subtask can suspend the current Smalltalk process or the Smalltalk parent task itself. Some Subtask class instance variables are used in implementing this system.

- brokenPipesProcess is a Process that runs continuously and forks error notifiers if it receives signals.
- scheduledSubtasks is a Dictionary containing all the current subtasks keyed by taskID. Subtasks are added to this list when they are started.
- unscheduledSubtasks is a Dictionary containing unscheduled subtask termination information keyed by taskID. This information, in the form of information from an executed wait system call, is collected and saved by the subtask management system. This information is collected when a task dies, and it is not recorded in the scheduled task list.
- scheduledSubtasksAccessProtect is a Semaphore for mutual exclusion used to protect accessing of the ScheduledSubtasks dictionary.
- waitProcess is a Process that runs continuously. Each time a 'dead child' signal is received, a wait system call is made, and the subtask management information is updated.

The subtask management system maintains a list of all currently active subtasks in ScheduledSubtasks.

Here is what happens when the Subtask wait message is used. When the system receives a signal that a child task has terminated (a "dead" child signal), the system performs a *wait* system call. The *wait* system calls returns the taskID of the dead child task. Usually, a *wait* system call is not made until a dead child signal is received so the Smalltalk parent task is not held up unnecessarily. The *wait* system call enables the system to update the ScheduledSubtasks dictionary. If a Smalltalk process is suspended via a Subtask wait message, the appropriate subtask's semaphore is signaled and the suspended Smalltalk process resumes.

Instead of having the parent Smalltalk task share the CPU with the spawned child task, competing for CPU cycles, you may want to block the parent task entirely. From waitWithSmalltalkSuspended, a *wait* system call is issued directly, which bypasses the dead child monitoring process. *wait* system calls, each of which suspends an entire parent task, are executed until the desired child task terminates. ScheduledSubtasks is updated after each *wait* system call, which means that both types of suspending – entire task blocked and single process suspended – can occur at the same time.

Applications – A CFileModel Example

Here is a simple example of a useful operating system interface application. This application allows you to compile C programs from within Smalltalk. Ways to expand this example are given after it is described.

You will note that this involves just a few methods and defines a class in category Interface-File Model. The class CFileModel is a subclass of FileModel and adds the class variable name CTextMenu to the class variables of FileModel.

Here is the class definition:

```
FileModel subclass: #CFileModel
  instanceVariableNames: ''
  classVariableNames: 'CTextMenu '
  poolDictionaries: ''
  category: 'Interface-File Model'
```

Here are the instance methods:

```
textMenu
  "Answer the menu for this pluggable MVC."
  "CFileModel flushMenus"

  fileName == nil ifTrue: [↑nil].
  CTextMenu == nil ifTrue: [CTextMenu ← ActionMenu
    labels: 'again\undo\copy\cut\paste\do it\print it\compile\put\get'
      withCRs
    lines: #(2 5 8 10 )
    selectors: #(again undo copySelection cut paste doIt
             printIt compile:from: accept getNew:from: )].
  ↑CTextMenu


compile: fullText from: controller
  "Compile the contents of the model, and notify the user of the result."

  | tempFile errors |
  tempFile ← FileStream fileNamed: '/tmp/stC',
      (Time millisecondClockValue printString), '.c'.
  tempFile nextPutAll: (controller paragraph asString).
  tempFile close.
  errors ← OS executeUtility: '/bin/cc' withArguments:
      (OrderedCollection with: tempFile fullName).
  tempFile remove.
  errors size > 0
      ifTrue: [self notify: errors]
      ifFalse: [self notify: 'No Errors']
```

Here are the class methods:

```
flushMenus
  "CFileModel flushMenus"

  super flushMenus.
  CTextMenu ← nil


open: aFileName
  "Use inherited methods to open a compilable view on a file. "
  "CFileModel open: 'sampleProgram.c'"

  self open: (self fileStream: (Disk file: aFileName))
    named: aFileName
```

You can open a view on an existing file or on a new file. Execute this expression to open a view on a new file:

```
CFileModel open: 'hello.c'
```

This opens a text window on an empty file. Type in the following C program (or choose your own C program):

```
#include <stdio.h>

 main()
  {
  printf("hello, world\n");
  }
```

Bring up the middle button menu and select compile. This runs the C compiler on the contents of the window. Once you have the contents as desired, the file can be written using the put menu item. After compilation, you should receive a confirmation notifier that the compile was successful, or one specifying your syntax errors. You can spawn a shell with OS shell if you like to verify that your program runs outside the Smalltalk system.

As noted earlier, this application implements a very simple C compilation environment, but you can certainly expand this application beyond its limited capabilities. Here are some suggestions for expansion:

- Make a separate view for syntax errors.
- Allow library specification and other compiler arguments.
- Spawn views of include files.

Refer to the Model-View-Controller section of this manual for information about views. Browse in the fileIn directory for other applications that create and manage views.

An Extension – OSFilter Class

The class OSFilter is a straightforward, simple extension of the Smalltalk OS interface. OSFilter is modeled on the Unix concept of a filter program. A filter accepts standard input and transforms the input, character by character, to make the output. The UTek commands tr and sed are examples of filters.

OSFilter uses the class Subtask to create the process that runs a standard UTek filter command or a filter program you have written. See the description of OSFilter in the *Tektronix Smalltalk Reference* manual for two examples of how to use OSFilter.

Section 9

Fonts in Smalltalk

INTRODUCTION

This section consists of three parts: Introductory Information, *Handling Fonts in Smalltalk*, and *Fonts Background.*

- The first part, *Introductory Information,* describes font terminology as it is used in Tektronix Smalltalk.
- The second part, *Handling Fonts in Smalltalk,* describes how to perform common tasks with fonts. It assumes that you know how Smalltalk manages fonts, how strings are associated with fonts, what mechanisms underlie the display of strings, etc. If this section does not give you enough understanding to use fonts in your programming efforts, go through the third part, the tutorial *Fonts Background.*
- The third part, *Fonts Background,* is an informal tutorial which guides you through those parts of the system that use font information. You will examine each of the major classes involved with fonts and text. You will see also how font information is associated with on-screen text. Go through this section first if you have not yet explored the font-related classes.

For a reference-oriented approach to fonts, see the classes StrikeFont, VirtualStrikeFont, StrikeFontManager, TextStyle, and TextStyleManager in the *Tektronix Smalltalk Reference* manual. Also, look in this reference manual for font charts describing Pellucida[1] family fonts.

INTRODUCTORY INFORMATION

The Font Directory

The standard Smalltalk image as it is shipped from the factory has a basic repertoire of fonts stored in it, or *virtually* stored in it. (VirtualStrikeFont allows fonts to be "referenced" in the system but not stored in it. VirtualStrikeFonts are read in only when they are to be displayed.) The sources for these fonts are the font files in the font directory, */usr/lib/fonts*. When you want to install new fonts in the Smalltalk image, you read the font files in this directory.

1. Pellucida is a registered trademark of Bigelow and Holmes.

Font Terminology Definitions

Typography has had a long history before the invention of computers. Because of this, there has not always been strict agreement of terms among the practitioners in the field. When computer technology was applied to typography, a refinement of some common terms and invention of others was necessary for the computer simulation of typographical information. What is presented here is the way that Tektronix Smalltalk approaches typographical information. Thus, some key terms are defined here so that there will be a common basis of understanding for what follows in this section. In particular, note that *typeface* is the more common term for *family*, which is used here. You will find, however, that most other terms are similar in definition to those used in common typographical practice.

font	A collection of typographical properties that apply to the graphical entities that represent (printable) characters in Smalltalk. The primary properties of a font are family, face, and size.
family	Family refers to the basic look of a set of characters that makes it distinguishable from another set. Family is the intrinsic property of a font. Families are named and frequently protected by copyright. Examples include "Helvetica", "Times Roman", and "Pellucida".
face	Face is the emphatic property of a font. Examples include Basal (no emphasis, or *regular*), Bold, Italic, BoldItalic, and Underlined.
size	Size is the dimensional property. It is typically specified by the height of capital "A" in points (72nds of one inch), although such a measure is more meaningful on paper than on a display.
StrikeFont	StrikeFont is the Smalltalk class that represents the abstract idea of a font defined earlier. An instance of StrikeFont represents a single combination of family, face, and size, with a bitmap for each ASCII character. (An instance of a StrikeFont is analogous to a complete set of characters in one typeface sitting in a printer's type box.) The Tektronix Smalltalk system is supplied with a number of font files that can each create an instance of StrikeFont, when read into the Smalltalk image.
VirtualStrikeFont	VirtualStrikeFont is similar to StrikeFont in all respects except that a VirtualStrikeFont is loaded into the system only when it is "referenced". At this time, it is turned into a StrikeFont. An instance of VirtualStrikeFont represents a single combination of family, face, and size, with a bitmap for each ASCII character.
TextStyle	TextStyle is a collection of instances of StrikeFonts. Usually, an instance of TextStyle is composed of related instances of StrikeFonts. For example, a TextStyle called "PellucidaSans-Serif10 and 12" installed in the standard image has fonts composed of the font files (basal, bold, italic, and underlined) 10 point size and 12 point size — some 16 instances of StrikeFont in all.
StrikeFontManager	StrikeFontManager is a dictionary of instances of StrikeFonts (or VirtualStrikeFonts). The central repository of instances of StrikeFonts loaded from font files into the image is the single instance of StrikeFontManager called the FontManager. The standard image as

shipped by Tektronix already has a basic set of fonts loaded into the FontManager. You can load other fonts from the font directory or delete fonts in the FontManager.

TextStyleManager	TextStyleManager is a dictionary of instances of TextStyle. The central repository of instances of TextStyle installed in the image from collections of font files is the single instance of TextStyleManager called the StyleManager. You can install new instances of TextStyles in the StyleManager by creating a name for a new TextStyle instance and specifying those font names that will make up the new TextStyle instance.

More information about these terms is found in the discussion of common font handling tasks in the *Handling Fonts in Smalltalk* part of this section.

HANDLING FONTS IN SMALLTALK

This discussion of fonts is task-oriented. You are given here suggestions about how to accomplish tasks commonly done with fonts. Look for a task that is similar to what you want to do to get some idea of how to accomplish it. Then explore the code and classes involved to find a solution to your programming task.

If nothing here seems to help, try the Smalltalk Reference manual or the Fonts Background tutorial later in this section. The general progress of examples is from conceptually simple to more difficult as you go on. Also, the System Workspace font information is explored first; then other font manipulation tasks are treated.

Fonts in the System Workspace

Inspecting Resident StrikeFonts

Go to the System Workspace and scroll down to about half way till you see the heading *Fonts and Text Styles*. Highlight FontManager inspect and then doit. Since the FontManager is a dictionary this brings up a dictionary inspector. Reframe the inspector if you need to so that you can see the entire list of key names in the lefthand pane of the inspector. These names are taken directly from the font file names in the font directory. The numbers indicate point size and the I, B, and U indicate italic, bold, and underlined. (No tag letter indicates basal – plain – font.)

Click on any key name, say PellucidaSerif10U, and note that its associated value is StrikeFont name PellucidaSerif10U emphasis 4. This is an instance of the StrikeFont named PellucidaSerif10U. This and all the other names you see have already been loaded into the StrikeFontManager for you at the factory. See later for adding more StrikeFonts to the FontManager.

Inspecting Resident TextStyles

Look under *Fonts and Text Styles* and find StyleManager inspect and then doit. This opens a dictionary inspector just like the FontManager did since they are both dictionaries. In the standard system, you will find a number of text styles already installed that show up in the left pane:

```
Pellucida Default 08 and 10
Pellucida Default 10 and 12
Pellucida San-Serif 10 and 12
Pellucida San-Serif 12 and 14
Pellucida San-Serif 18 and 24
Pellucida Serif 10 and 12
Pellucida Serif 12 and 14
Pellucida Serif 18 and 24
Pellucida TypeWriter 10
Pellucida TypeWriter 12
```

These names are arbitrary in the sense that you the programmer supply these names when you create new text styles. (You might guess from the inclusion of spaces that these names are not taken directly from font file names like StrikeFont names are.)

Do a middle button inspect of Pellucida Default 10 and 12 to open an inspector on the text style itself. Click on the instance variable fontArray to see the array of StrikeFonts included in this particular text style. You can "mix and match" any fonts you like in your own text style but usually StrikeFonts of the same family are chosen.

Installation of a new TextStyle

Look under *Fonts and Text Styles* and find the following code:

```
StyleManager
        styleName: 'Pellucida Sans-Serif 8 and 10'
        baseNames: #('PellucidaSans-Serif8' 'PellucidaSans-Serif10')
        lead: 3.
```

This is the code you use to install new text styles in the StyleManager. In the System Workspace, select this code and doit. Now go back to the StyleManager inspect and doit. You have just installed a new TextStyle called Pellucida Sans-Serif 8 and 10.

Look at the code for a minute. The styleName: 'Pellucida Sans-Serif 8 and 10' part of the method specifies the user-supplied (and completely arbitrarily named) text style name. The baseNames: #('PellucidaSans-Serif8' 'PellucidaSans-Serif10') part of the method is a little more interesting. If you look in the font directory, you will observe that there are a lot of font names among which are *PellucidaSans-Serif8.font, PellucidaSans-Serif8I.font, PellucidaSans-Serif8B.font,* and *PellucidaSans-Serif8X.font*, the I, B, and X meaning italic, bold, and extra bold. The installation of text styles has been simplified somewhat by introducing the idea of a *basename* for a font file name. Methods using basenames for font files allow you to leave off the I, B, or X from the font name. The lead: 0 part of the method specifies the distance in pixels between lines of characters. See Figure 9-1, *Installing a Text Style.*

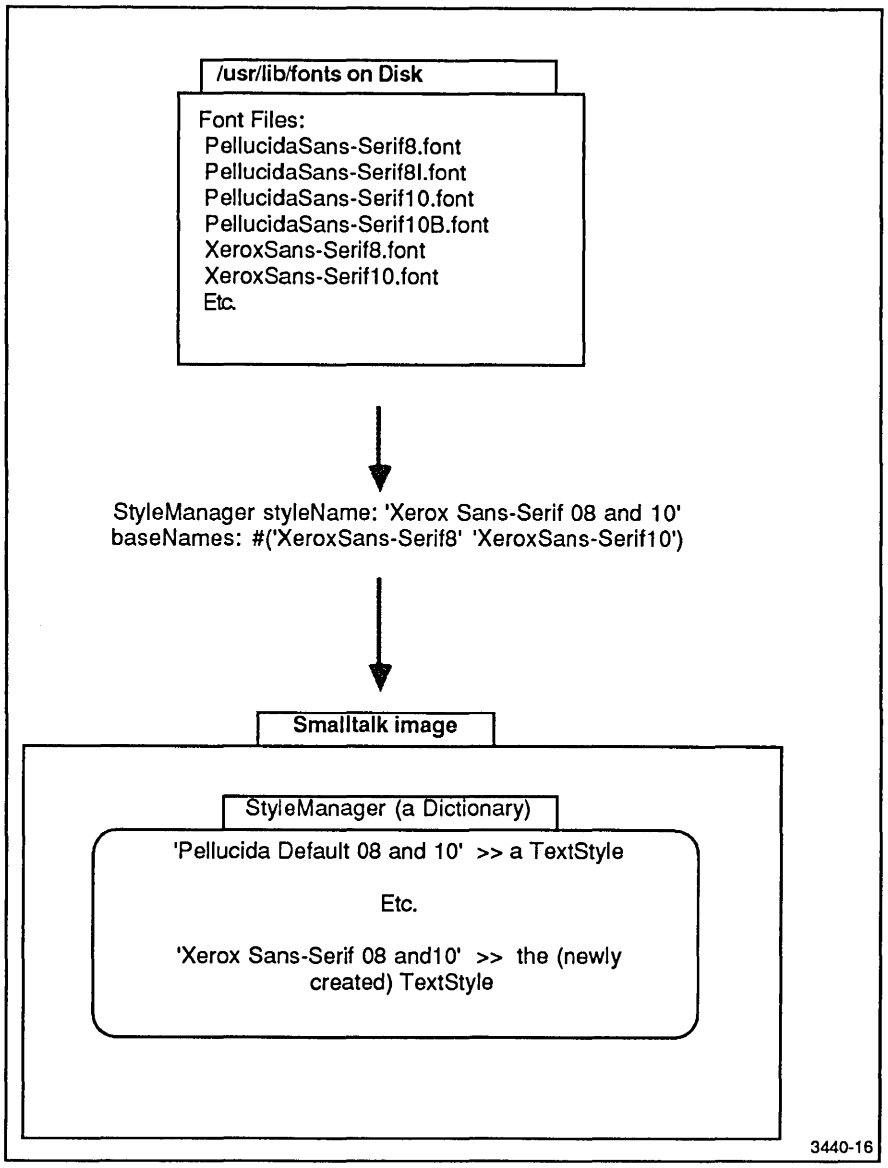

/usr/lib/fonts on Disk
Font Files:
PellucidaSans-Serif8.font
PellucidaSans-Serif8I.font
PellucidaSans-Serif10.font
PellucidaSans-Serif10B.font
XeroxSans-Serif8.font
XeroxSans-Serif10.font
Etc.
StyleManager styleName: 'Xerox Sans-Serif 08 and 10' baseNames: #('XeroxSans-Serif8' 'XeroxSans-Serif10')
Smalltalk image
StyleManager (a Dictionary)
'Pellucida Default 08 and 10' >> a TextStyle
Etc.
'Xerox Sans-Serif 08 and10' >> the (newly created) TextStyle
3440-16

Figure 9-1. Installing a Text Style.

Choosing a Default Text Style

Look under *Fonts and Text Styles* and find the following code:

```
StyleManager changeDefaultTextStyle
```

Executing this code allows you to easily choose a new default text style for the system. You will see that you have a choice of a number of text styles. Choose a new one from the menu. You should see the windows redraw themselves with the new text style installed. You now have a new default text style. This new text style is the default style because it has been installed in the pool dictionary TextConstants as a variable called DefaultTextStyle. Various kinds of classes know about DefaultTextStyle. Some of these classes are: Text, StrikeFontManager, ParagraphEditor, DisplayText, Paragraph, StrikeFont, and TextStyle. So, you can see that just about everything having to do with text uses DefaultTextStyle.

If you like, you can open an inspector on the pool dictionary TextConstants to see the variable DefaultTextStyle and many other related variables.

Creating a StrikeFont

If you simply want to create a StrikeFont out of a font file, you can use the following expression:

```
aNewSF ← StrikeFont readFrom: (FontManager fontFileStream:
'PellucidaSans-Serif10')
```

fontFileStream: returns a FileStream on the specified font file. See Figure 9-2, *Creating a StrikeFont.*

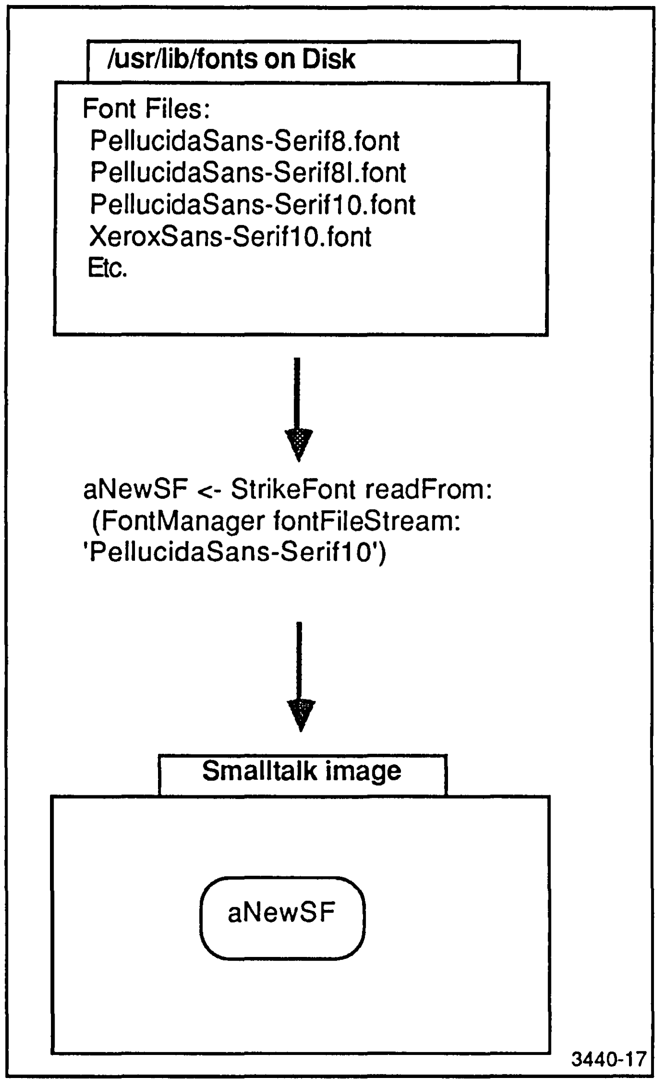

/usr/lib/fonts on Disk
Font Files:
PellucidaSans-Serif8.font
PellucidaSans-Serif8I.font
PellucidaSans-Serif10.font
XeroxSans-Serif10.font
Etc.
aNewSF <- StrikeFont readFrom: (FontManager fontFileStream: 'PellucidaSans-Serif10')
Smalltalk image
aNewSF
3440-17

Figure 9-2. Creating a StrikeFont.

Installing a Font in the FontManager

Usually, the installation of fonts in the FontManager is accomplished during the installation of a text style in the StyleManager, but there may be times when you want to install an individual font directly into the FontManager itself. Perhaps, you don't want this font to be associated with a text style since you intend to work with just the individual font by itself. You can send the install: message to StrikeFontManager to install a StrikeFont in the FontManager. Try these expressions to see how this is done.

```
FontManager install: 'XeroxSans-Serif8'.
FontManager inspect
```

Note that the file name string is taken from the font directory, */usr/lib/fonts*. Open a file list on */usr/lib/fonts* to see the font file names. See Figure 9-3, *Installing a StrikeFont in the FontManager.*

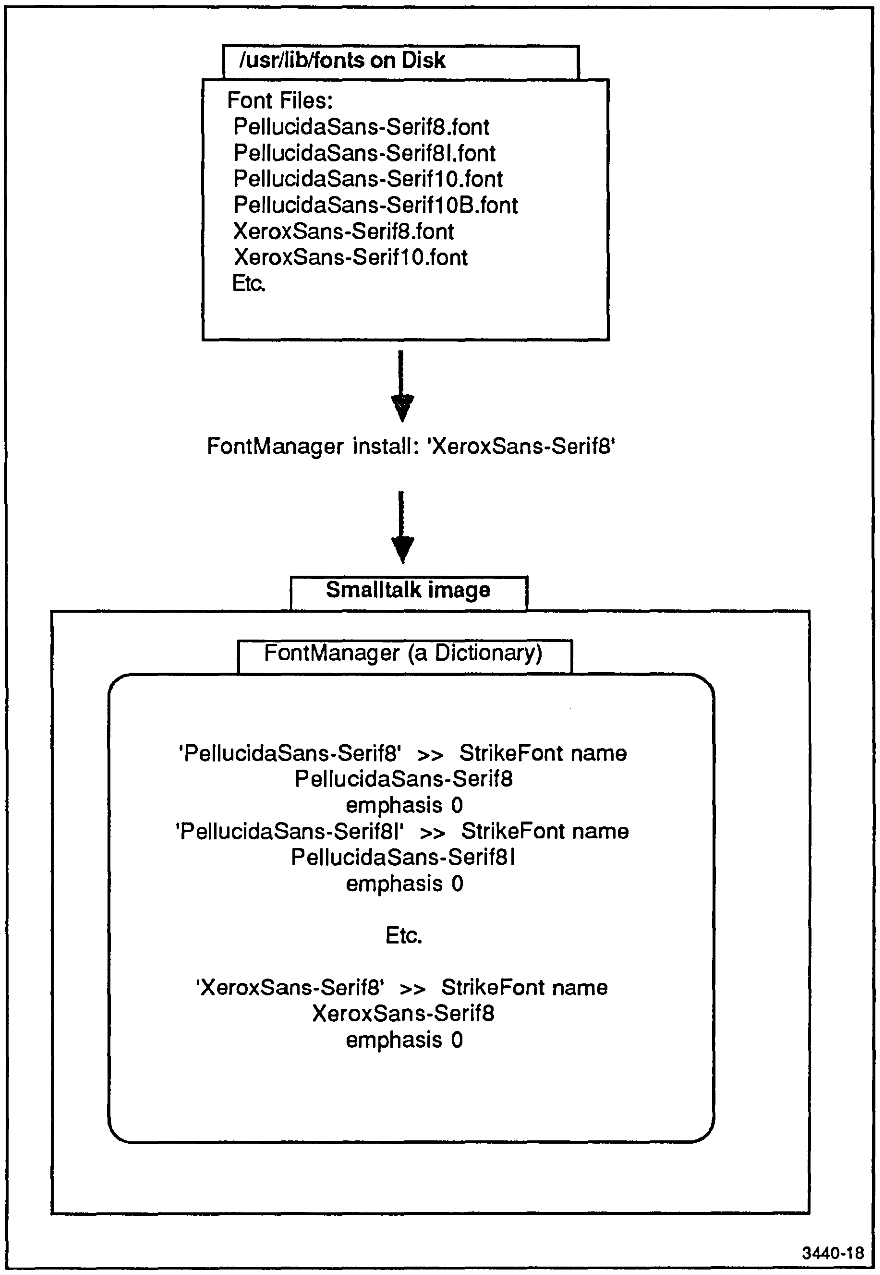

/usr/lib/fonts on Disk
Font Files:
PellucidaSans-Serif8.font
PellucidaSans-Serif8I.font
PellucidaSans-Serif10.font
PellucidaSans-Serif10B.font
XeroxSans-Serif8.font
XeroxSans-Serif10.font
Etc.
FontManager install: 'XeroxSans-Serif8'
Smalltalk image
FontManager (a Dictionary)
'PellucidaSans-Serif8' >> StrikeFont name PellucidaSans-Serif8 emphasis 0
'PellucidaSans-Serif8I' >> StrikeFont name PellucidaSans-Serif8I emphasis 0
Etc.
'XeroxSans-Serif8' >> StrikeFont name XeroxSans-Serif8 emphasis 0
3440-18

Figure 9-3. Installing a StrikeFont in the FontManager.

How to Display Unprintable Characters

Some Tektronix fonts have printable characters below ADE (ASCII decimal equivalent) 32 and above ADE 127 that cannot be accessed through the keyboard. For example, Tektronix Pellucida Serif and Pellucida Sans-Serif font have only 127 characters, but Pellucida TypeWriter have 255 characters. The access to these is a little roundabout, but you can print them on the screen with the following code:

```
aString ← String new: 255.
1 to: aString size do: [:characterIndex | aString at: characterIndex put:
  (Character value: characterIndex)].
aString asDisplayText display
```

Note that a Character is placed in a String and then converted to DisplayText so that it can be displayed.

Displaying Fonts

Displaying a StrikeFont

Remember that a "font" in Smalltalk is an instance of StrikeFont. So, how would you go about taking a look at some of the fonts that you know are resident on disk in font files in the font directory or resident in the FontManager?

Open an inspector on the FontManager and take a look at the font names in the left pane and pick out one, for example, PellucidaSerif8. Now, using this font name, execute the following expression:

```
anSF ← FontManager at: 'PellucidaSerif8'.
```

This extracts a StrikeFont from the FontManager and assigns it to anSF. If you don't see the StrikeFont you want in the FontManager, then you need to use the fontNames: method to install it. (Note that the install: method is used earlier to accomplish a similar task.) First, open a file list inspector on the font directory to see what font file names are available to you. Pick one out, say, *PellucidaSerif18.font*, a relatively large size font. Omit the *.font* file extension from the font file name and execute the following expression:

```
FontManager fontNames: #( 'PellucidaSerif18' )
```

Take another look at the FontManager with an inspector. PellucidaSerif18 should be there. This time extract the newly loaded StrikeFont with:

```
anSF ← FontManager at: 'PellucidaSerif18'.
```

You are ready finally to view some characters from this StrikeFont on the screen. Execute the following expression:

```
anSF displayLine: '12345678' at: Sensor waitClickButton.
```

You should see the string "12345678" appear at the cursor location when you press the left mouse button. Try putting in different strings, if you like, to see what the various characters look like. Try loading different fonts from the font directory, if you like.

Here is a "quick and dirty" way to view all of the characters in a StrikeFont (if the StrikeFont size is not too large). Execute the following expressions:

```
FontManager fontNames: #( 'Micro5B' ).
anSF ← FontManager at: 'Micro5B'.
anSF glyphs displayAt: Sensor waitButton.
```

The glyphs method in the last line of the code extracts that part of a StrikeFont that is the actual form holding the bitmap information for each character in a StrikeFont. This form is then displayed on the screen.

Note that you can remove a StrikeFont, just like any key in any dictionary, with the expression: FontManager removeKey: 'fontName'.

Extracting a Character Form

For some applications, you may want to extract just one character from a StrikeFont so you can treat it as a form, and perhaps use the bit editor on it. The message characterForm when sent to a StrikeFont extracts a character form from the glyphs form that contains all the characters in a StrikeFont.

The following code shows you one way to manipulate character forms. The first expression chooses one of the StrikeFonts in the default text style in the StyleManager. (See earlier under this heading for more about the StyleManager.) Note that the pool dictionary TextConstants holds the default text style associated with the symbol DefaultTextStyle. The default text style contains an array of StrikeFonts, the first StrikeFont of which becomes the default StrikeFont used by the system.

In the first expression of the following code, you are asked to specify which StrikeFont you would like to see by typing in a number from 1 to 9. These numbers correspond to the StrikeFonts in the fontArray in the default text style. When you type <Ctrl-1>, <Ctrl-2>, etc., at the keyboard while you have text selected in a text window, you see the text cycle through the StrikeFonts in fontArray in the default text style. This is simply another way of getting at the same thing.

The second expression asks you to supply a character whose form you would like to extract. The third expression displays this form on the screen at a place chosen by you.

```
anSF ← (TextConstants at: #DefaultTextStyle) fontAt:
        (FillInTheBlank request: 'Type in a number from 1 to 9:'
        initialAnswer: '1') asNumber.
aCharacterForm ← anSF characterForm: ((FillInTheBlank request:
        'Type in one character:' initialAnswer: 'a') at: 1).
aCharacterForm displayAt: Sensor waitClickButton.
```

Perhaps you would like to bit edit this character form. Try the following expression.

```
BitEditor openOnForm: aCharacterForm.
```

Changing the Font of a String

This example is written to show you how fonts can be applied to a string. It will allow you to explore the fonts in the FontManager so that you can get a quick idea of how the various fonts look.

The first expression uses a FillInTheBlank to collect a string from you. The second expression returns a sorted collection of the font names currently available to you from the FontManager. (See earlier for how to add new StrikeFonts to the FontManager.) The next three expressions put the font names in a Stream so that in the next expression a PopUpMenu can display the font names as labels in a menu. The following expression returns your choice of font, and the last expression puts it up on the screen at the cursor position.

```
aStringOfText ← FillInTheBlank request: 'Type in a string of text;'
        initialAnswer:'abcdefghijklmnopqrstuvwxyz'.

"Go get font names from FontManager — it's a dictionary."

strikeFontNames ← FontManager keys asSortedCollection.

"Create a writable Stream as a String to put font names in."

aStream ← WriteStream on: (String new: 1024).

"Gather the font names and put them in aStream."

strikeFontNames do: [:aString | aStream nextPutAll: aString; cr].
aStream skip: -1.

"Make a PopUpMenu and install the font names as labels so you
  can select them."

strikeFontMenu ← PopUpMenu labels: aStream contents lines: nil alignment: 0.

"Start up the PopUpMenu and return the font name you have picked."

(i ← strikeFontMenu startUp) > 0
        ifTrue: [aStrikeFont ← FontManager at: (strikeFontNames at: i)]
        ifFalse: [↑nil].

"Now display the string in the chosen font."

aStrikeFont displayLine: aStringOfText at: Sensor waitClickButton.
```

Changing the Emphasis in Some Text

Instances of Text have two instance variables: string, which is a String holding ASCII character information and runs, which is a RunArray containing integers (one for each character) that represent emphasis. (See the tutorial exploration of Text and other font-related classes later in this section for more information.)

First, put a string of text on the display with:

```
t ← 'ABCDEFGHIJKL' asText.
t asDisplayText displayAt: Sensor waitButton
```

Now, change the emphasis codes of the instance of Text with the following expressions. Note that with the method emphasizeFrom:to:with: you can change the emphasis of individual runs of characters in the string. Here it is used to change the entire string of characters. When you run this code, it starts at 1 and increments emphasiscode until you do a <Ctrl-c>. After a while you will not see a change in emphasis. This means that you have run out of StrikeFonts in the default text style to display. These emphasis codes correspond to the <Ctrl-1>, <Ctrl-2>, <Ctrl-3>, etc., that you can do in a workspace on selected text. Try it and see. However, you cannot go beyond <Ctrl-9> in the workspace.

```
t ← 'ABCDEFGHIJKL' asText.
emphasiscode ← 1.
[true] whileTrue:
        [t emphasizeFrom: 1 to: (t string size) with: emphasiscode.
        t asDisplayText displayAt: Sensor waitClickButton.
        emphasiscode ← emphasiscode + 1]
```

FONTS BACKGROUND

As you go through this informal tutorial, be sure to use your Smalltalk system to open the inspectors, browse in the System Browser, execute the code, etc.

Introduction

Since text arranged on a page in a particular typeface involves the complexities of the typesetter's art and science, you might expect Smalltalk's emulation of this to involve a number of layers, which working in concert give you text in a typeface arranged in paragraphs in a window. And, sure enough, if you start looking around in the System Browser for font, text, and paragraph kinds of things, you will discover that the following classes all play their part: Character, String, Text, DisplayText, Paragraph, StrikeFont, TextStyle, StrikeFontManager, and TextStyleManager. (There are others, but this tutorial will concentrate on these.)

Probably the best way to understand how Smalltalk displays font information as a part of text within Smalltalk windows is to start with Characters and Strings. You will then see how font information is added to ASCII character and string information, and then, how this is put together as paragraph information, which is then displayed within windows. So, let's start with the classes Character and String.

Characters and Strings

The way Smalltalk handles character and string information is similar to but not the same as the way other languages do, such as C and Pascal. If you look at Character initialize in the System Browser, you will find that this method creates a table in which ASCII character values are associated with the alphanumeric symbols you are familiar with. (Smalltalk does, however, extend the range of characters beyond the ASCII table to 256.)

Strings are indexed collections of Characters, as the comment in the System Browser says. Note, especially, that an instance of String is simply an indexed collection of Characters — no less and no more. Looking further in the System Browser, you will see that you can compare strings in various ways, access the Characters in a String in various ways, and copy Strings. The methods in these message categories manipulate Strings in familiar ways.

An important point to understand in the beginning is that strings do not know how to display themselves. That is, Strings must appeal to some other class to display themselves. As you probably know, you can send the message display to any object you suspect might be able to display itself, and many times, the object displays itself in the upper right hand corner of the screen. However, Strings cannot do this. But, stop a moment and look at the two message categories displaying and converting.

Look at String displayAt:. The method's code looks like this:

```
self asDisplayText displayAt: aPoint
```

Note from this expression that a String must first be associated with a displayable object (DisplayText) before it can be displayed. Now look at the message category converting. Here you will find a number of asXXXX message selectors. The "conceptual" hierarchy leading to the display of text in windows is this: asText, asDisplayText, and asParagraph. Each of these methods associates more information with a String as you go from asText to asParagraph. Let's look at asText first:

```
↑Text fromString: self
```

This method apparently associates a String with a Text object, so let's look at Text now.

Text

Smalltalk requires that Strings somehow display themselves. A step on the way to this is the Text object. In the System Browser, you can see that the class Text has two instance variables: string and runs. string turns out to be a String, and runs turns out to be a RunArray of Integers that represent the emphasis of each Character in the Text object. Emphases are bold, italic, underlining, a change to a new typeface, etc. These are what you invoke when you do <Ctrl-1>, <Ctrl-2>, <Ctrl-3>, etc., in a workspace or some other text window. (If you haven't tried this before, select some text in a workspace and try <Ctrl-2>, <Ctrl-3>, etc., and then return to <Ctrl-1> — plain text — when you are done.) So, with runs you are starting to get at the displayable characteristics of text, but you are not quite there yet. The emphasis codes in runs allow you to invoke those displayable characteristics, though.

Note that Text has a message category converting, but not displaying (as String did). Within converting, you will find the familiar asXXXX message selectors. Thus, you find asDisplayText and asParagraph. In standard Smalltalk, Texts need to be converted to DisplayTexts to display themselves. Consequently, Strings and Texts both need display information associated with them, usually via DisplayText or Paragraph, before they can be associated with font information and displayed. Of course, when you are interacting with Smalltalk code and text in a workspace or the System Browser, Strings and Texts are converted to DisplayTexts or Paragraphs so that they can be displayed. Let's look at DisplayText next where finally you find displayable information.

DisplayText

DisplayText is one more step up in the text-information hierarchy. Execute the following code to create an inpector on a DisplayText object.

```
'0123456789' asDisplayText inspect
```

Note that there are four instance variables: text, textStyle, offset, form. Taking a look at text shows you that text is a Text object. Since this is now familiar, take a look at textStyle. In fact, do an inspect from the middle button menu to open up an inspector on textStyle. You can see from the instance variable names in the TextStyle inspector that you have (at long last) found where String and Text display information is. As you select each instance variable (fontArray, lineGrid, baseline, etc.), observe that most of the variables hold integer values. However, come back to fontArray. It is evidently an Array, but an Array of what? The elements of this array are "StrikeFont name PellucidaSans-Serif8 emphasis 0", "StrikeFont name PellucidaSans-Serif8B emphasis 0", and so forth. This is not immediately obvious until you do an inspect on the fontArray instance variable.

If you have done some browsing in the Graphics-Support category of the System Browser, you may have noticed that StrikeFont is a class name. Also, "PellucidaSans-Serif8", you might guess as some sort of typeface name, and, if you have some previous acquaintance with typesetting you might also figure out that the "8" means 8 point size type. "emphasis" means the specific ways a particular typeface can look such as bold, italic, bold italic, and underlined.

One more inspector will bring out all the major classes involved in how a string gets displayed and where it gets that display information. If you now inspect one of the elements in fontArray, you will be looking at the Smalltalk object that contains bitmap display data for ASCII characters. Note that, like TextStyles, StrikeFonts are moderately complex, but for now concentrate on four instance variables: xTable, glyphs, name, and emphasis. Let's dispense with the last two variables first. name is a String holding the name of the typeface and emphasis holds an index into fontArray, which is an instance variable of TextStyle. fontArray holds StrikeFonts.

However, take a moment to look at glyphs. It turns out to be a form that holds the bitmap information for each ASCII character in a particular typeface, emphasis, and size. You can think of this form as a long form, one character high, holding the characters in ASCII order from left to right. Since you need to extract the part of glyphs that corresponds to each individual ASCII character, you need to know how to locate the place of each character in the form. xTable is the instance variable that holds this data. The elements of the xTable integer array are the left x-coordinate of the "box" surrounding each character in the form. To correctly index into xTable to find the left x-coordinate of a character's form, use the character's asciiValue + 1 like this:

```
xTable [aCharacter asciiValue + 1]
```

See Figure 9-4, *The* glyphs *form.*

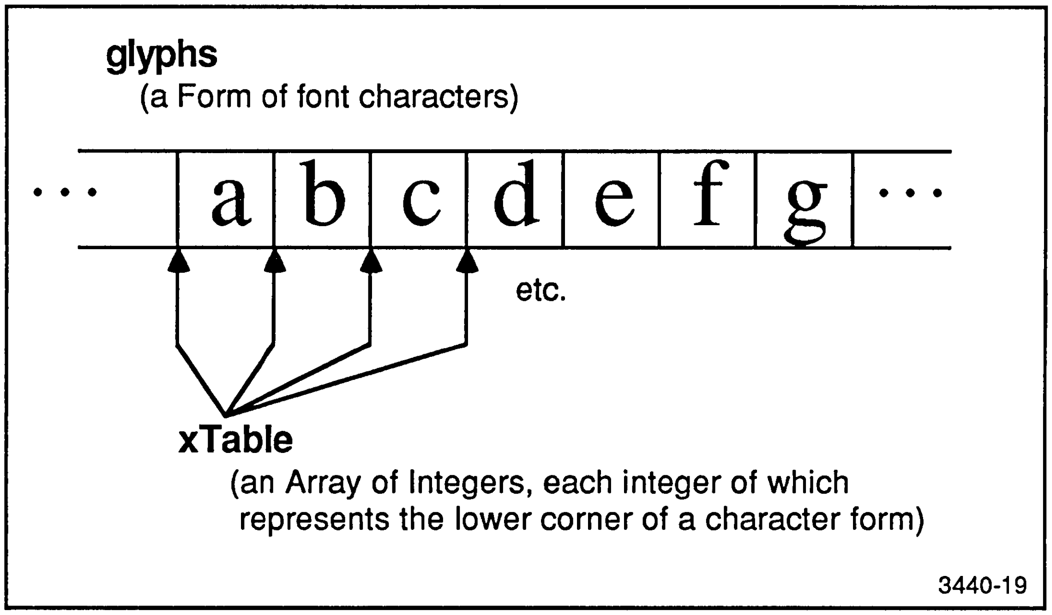

glyphs
(a Form of font characters)
... a b c d e f g ...
etc.
xTable
(an Array of Integers, each integer of which represents the lower corner of a character form)
3440-19

Figure 9-4. The glyphs Form.

Paragraph

Introduce yourself to the class Paragraph now by executing the following expression in a workspace:

```
'0123456789' asParagraph inspect
```

This opens an inspector on the string '0123456789' converted to a Paragraph. The primary difference between DisplayText and Paragraph as you can see from a comparison of the instance variables in their respective inspectors is the addition of more instance variables in the Paragraph. From the names of these additional instance variables, you can see that they deal with two additional aspects of text information. They add information about how to deal with text information as a complete form (clippingRectangle, compositionRectangle, destinationForm, rule, mask) and information about how to deal with text information as a paragraph (marginTabsLevel, firstIndent, restIndent, rightIndent, lines, lastLine, outputMedium). If these additional instance variables interest you at this time, take a look in the System Browser under Graphics-Display Objects, Paragraph, comment. It is sufficient here just to note that DisplayText is a simpler Paragraph, which is shown in the hierarchy in the System Browser.

Displaying StrikeFonts

Before you proceed to the task-oriented, Handling Fonts in Smalltalk, take some time now to look at and execute the following Smalltalk expressions.

Emphasis Codes

Here is some Smalltalk code that lets you step through the emphasis codes for the standard font. After you type this into a workspace and do it, you should <Ctrl-c> out of the loop. Note that these codes are the same codes that <Ctrl-1>, <Ctrl-2>, <Ctrl-3>, etc., change when you do this in a workspace or other text window.

From the discussion earlier you know that the Text object, t, has to be converted to a displayable object by the asDisplayText message in order to see it. Move the mouse around some while you click a mouse button to cycle through the emphasis codes. Note that after about 25 clicks or so the emphasis does not change any more. This number may be different when you change TextStyles.

```
emphasiscode ← 1.
[true] whileTrue: [t ← Text string:
    'this is a string' emphasis: emphasiscode.
t asDisplayText displayAt: Sensor waitClickButton.
emphasiscode ← emphasiscode + 1]
```

Some StrikeFont Display Code

The following code allows you to change and display StrikeFonts. Before you execute the example code, open a file list on the string returned by printing OS fontDirectory name. This gives you the names of different font files to type in response to the fill-in-the-blank request. Just type in the complete file name excluding the file suffix (*.font*).

The following code:

- Gets a font file name from you.
- Creates a StrikeFont from an externally stored font file. (And, as a part of this, the character bitmaps are created and stored in the glyphs instance variable.)
- Associates a TextStyle with the StrikeFont.
- Creates a Paragraph instance and specifies a TextStyle for it.

Finally, the Paragraph instance is converted to a displayable form, which is then displayed at the cursor coordinates when you click a mouse button. (The Cursor execute showWhile: code puts up the asterisk/arrow "busy" cursor.)

```
aFontName ← FillInTheBlank request: 'What font file?
  (Look in a File List at the font directory)?' initialAnswer:
  'PellucidaSerif10'.
aFileName isEmpty ifTrue: [↑self].
aStrikeFont ← StrikeFont readFrom: (FontManager fontFileStream: aFontName).
  aTextStyle ← aStrikeFont asTextStyle.
aParagraph ← Paragraph withText: 'abcdefghijklmnopqrstuvwxyz
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  1234567890' asText style: aTextStyle.
form ← aParagraph asForm.
Cursor execute showWhile: [form displayAt: Sensor waitClickButton.]
```

TextConstants

You have probably wondered as you read through this discussion where the default font information is stored in the image. To find and explore this, look now in the System Workspace. Find the heading, Globals; it is about half way down in the workspace. Note that along with other global variables, there are two with familiar looking names: FontManager – a StrikeFontManager and StyleManager – a TextStyleManager. See the discussion of these later in this section. For now, look further down the list to the heading Variable Pools (Dictionaries). Under this heading is something called TextConstants. Open an inspector on this. Explore this pool dictionary a little while, if you like, and then take a look at the key entry DefaultTextStyle.

Do a middle button inspect on DefaultTextStyle. The inspector shows you that DefaultTextStyle is a TextStyle instance. And, if you examine the instance variable fontArray, you will see the StrikeFonts available to you as defaults. Doing a references operation shows you where DefaultTextStyle is used. As you might expect, DefaultTextStyle is used to initialize the Paragraph and TextStyle classes. The expression Paragraph withText: shows you how default TextStyles are obtained from the system for Paragraphs. After this, take a look at Text, class initialization in the System Browser. This should give you a good idea how the class Text is initialized.

Conclusion

After you have gone through this guided tour of String, Text, DisplayText, and so forth, you should find the font information in *Handling Fonts in Smalltalk* easy to apply to your Smalltalk applications.

Section 10
The Interpreter

INTRODUCTION

This section documents the important features of the Tektronix Smalltalk interpreter. The most significant feature is the removal of the object space limit of the Smalltalk interpreter specified in the Addison-Wesley book by Adele Goldberg and David Robson.

This and other differences between that specification and the Tektronix Smalltalk interpreter are described here. The section is divided into a description of overall performance, followed by comparisons of CompiledMethod, MethodDictionary, and primitive methods (Smalltalk machine-language calls).

PERFORMANCE CHARACTERISTICS

The Tektronix Smalltalk interpreter has a number of useful characteristics for developing large Smalltalk applications. These are:

- The number of objects in the image are limited only by system memory.
- The size of objects are limited only by system memory.
- The Tektronix interpreter uses 32-bit object-oriented pointers (oops).
- SmallInteger values are in the range -2^{30} to $2^{30}-1$.

Object-Oriented Pointers

Oops (object-oriented pointers) are the values used by the interpreter to name objects. Of the 32 bits in an oop, only 29 are used to actually name objects. Thus, there is a theoretical maximum of about 500 million objects in the system. A practical limit for the maximum number of objects on the systems with the Tektronix interpreter depends on the average size of objects and your system memory configuration. With an average object size of 50 bytes, together with a practical object memory size of 6 to 10 megabytes, the system allows approximately 120,000 to 200,000 objects.

SmallIntegers

SmallIntegers in the Tektronix system are represented with 31 bits. Operations on integers representable by SmallIntegers use SmallInteger primitive operations. Operations on integers larger than 31 bits use code written in Smalltalk.

Table 10-1
Tektronix Smalltalk Interpreter Characteristics

Characteristic	Tektronix Smalltalk
SmallInteger Size	-2^{30} to $2^{30}-1$
Maximum Number of Objects	Memory Size Limited
Size of Byte Indexable Elements	8 bits
Size of Word Indexable Elements	16 bits
Size of Object Indexable Elements	32 bits
Maximum Size of Byte Indexable Objects	Memory Size Limited
Maximum Size of Word Indexable Objects	Memory Size Limited
Maximum Size of Object Indexable Objects	Memory Size Limited

TEKTRONIX INTERPRETER DESIGN

The Tektronix interpreter has some important characteristics relevant to its overall design. In the Tektronix system, object management has been considerably changed from the specification used by Goldberg and and Robson – the Tektronix system does not use an object table.

In addition, the class CompiledMethod is substantially different. CompiledMethod looks and behaves much more like other classes. MethodDictionaries also have a different structure.

The Object Table

In Goldberg and Robson chapter *Formal Specification of Object Memory*, the authors specify the structure of the object table. This 32K entry table restricts the maximum number of objects to the number of entries in the object table.

The Tektronix interpreter does not use an object table, since, with oops of 32 bits, the number of possible entries in such an object table would be impractical to manage. The benefits of eliminating an object table are:

- There is no *object-table* limit to the number of objects in the system.
- There is no extra level of indirection involved with every single operation on objects – creation, destruction, and manipulation.

There is, however, a penalty incurred with several rarely used methods, such as, become:. See *Primitive Methods* for a discussion of this.

CompiledMethods

Class CompiledMethod is consistent with the standard Smalltalk object structure. (The original CompiledMethod specified by Goldberg and Robson was non-standard, and required special treatment.) See Figure 10-1, *Structure of an Instance of CompiledMethod.* Instances of CompiledMethod consist of three objects: the CompiledMethod structure itself, an instance of LiteralArray, and an instance of ByteCodeArray. Note that the *Source Code Reference* field in the CompiledMethod structure contains a reference to the source code on disk.

This representation permits the creation of subclasses of CompiledMethod. Protocol for CompiledMethod formally support access to the source code reference. The source code reference in instances of CompiledMethod is divided into two fields. The three high order bits represent a zero-based reference into the global variable SourceFiles. This global contains an array of files. The remaining 27 low order bits in this SmallInteger represent the position of this method´s source code in the file referenced by the three high order bits.

This representation eliminates any need for special primitives for creating (newMethod:header:) and accessing components of (literalAt: and literalAt:put:) instances of CompiledMethod.

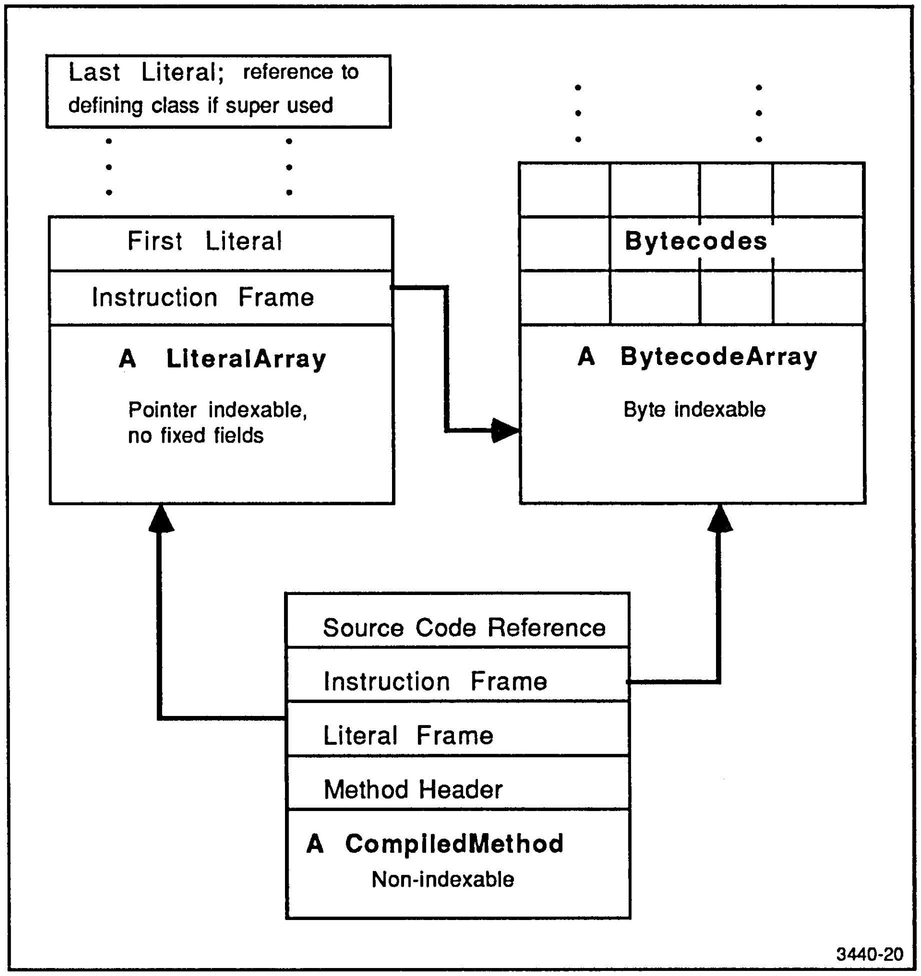


Figure 10-1. Structure of an Instance of CompiledMethod.

Method Dictionaries

The representation of instances of MethodDictionary is different from the Goldberg and Robson specification. See Figure 10-2, *Structures of Instances of MethodDictionary.*

Goldberg and Robson specify that instances of method dictionaries contain the keys (CompiledMethod selector names) as part of the method dictionary object itself. The Tektronix interpreter specifies that the keys, instead of being part of the MethodDictionary object itself, are contained in a separate Array object that holds the selector names.

Separation of the keys eliminates the use of become: . In the Goldberg and Robson specification, become: is used to accomplish an atomic update operation. Since become: is a relatively slow operation, the Tektronix interpreter accomplishes the atomic update by methods that rely on the separation of keys.

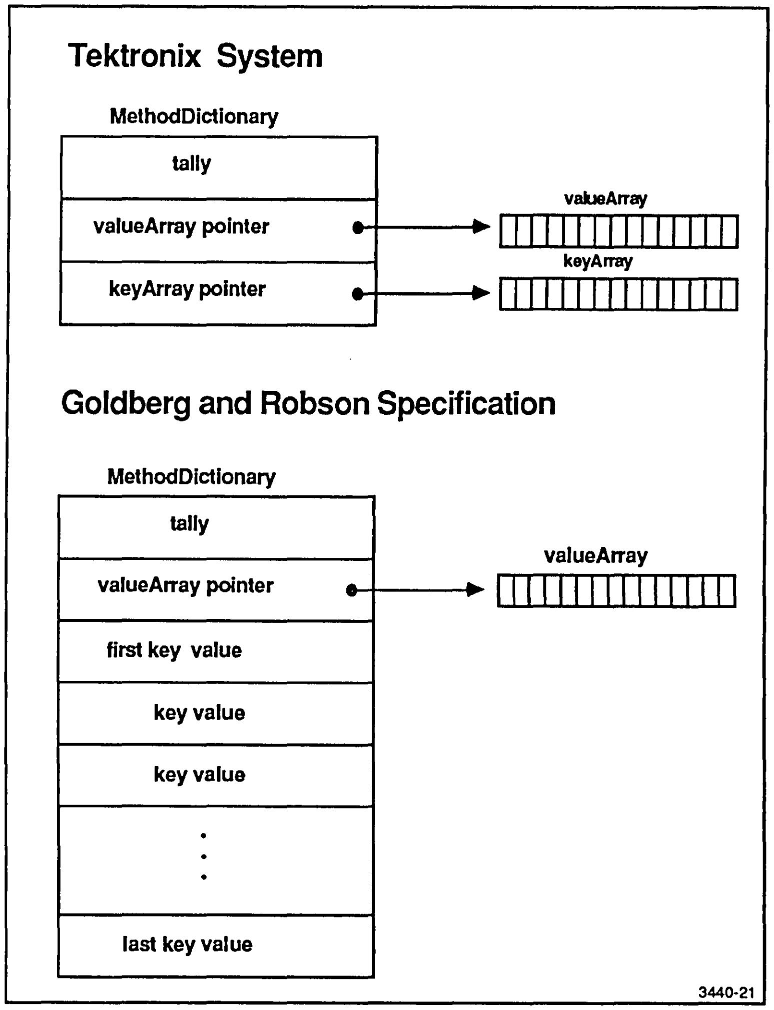


Figure 10-2. Structures of Instances of MethodDictionary.

PRIMITIVE METHODS

Some Smalltalk methods are implemented by making machine language calls directly. These methods are called primitive methods. In the Goldberg and Robson book, the chapter *Formal Specification of the Primitive Methods* gives a list of primitive methods along with their associated primitive indexes. In the Tektronix system, some of the primitive methods specified in this book are eliminated, others are added, while other methods function in a different way.

Primitives Not Implemented

A number of the primitives specified in the Goldberg and Robson book are not applicable to the Tektronix system and are not implemented.

21 – 37 The primitives in this range are defined to perform arithmetic on larger than 16-bit LargePositiveInteger objects. In the Tektronix interpreter, these primitives are not implemented since the Tektronix interpreter uses 31-bit wide small integer values. There is no meaningful distinction for most practical applications between the instances of SmallInteger and the instances of LargePositiveInteger and LargeNegativeInteger falling within the 31-bit limit. Thirty-one bit integer arithmetic operations are done at the machine language level. (Arbitrary precision arithmetical operations can still be done using Smalltalk methods for integers that cannot be expressed in 31 bits.)

68, 69, & 79 These primitives are not implemented, since they deal with the CompiledMethod of the Goldberg and Robson specification. The Tektronix system uses a different CompiledMethod.

76 The asObject primitive is not implemented in the Tektronix interpreter because there is no object table.

In the Goldberg and Robson specification, asOop and asObject function as inverses. Sending asOop to an object returns an integer representing its oop – the object table reference to the object. This value is typically be used as a hash code for the object. Sending asObject to an oop (represented by a SmallInteger) returns the object. Since there is a one-to-one correspondence between objects and object table reference values, you can be certain that two objects are the same object if they have the same asOop value (for a system based on the Goldberg and Robson specification).

The Tektronix interpreter does not use an object table, so the asOop method returns a value with a different meaning. This value is the hash value calculated for each object at its creation. These values are not guaranteed to be unique for every object. In practice, the vast majority of hash values are unique. Since there is no guaranteed unique asOop value for each object, there is no reason to use asObject. As a result of this, asObject is not implemented in the Tektronix system.

78 The nextInstance primitive is not implemented since there is no inherent ordering of objects in the Tektronix system. The Goldberg and Robson specification uses nextInstance in conjunction with someInstance to obtain all existing instances of some class. In the Tektronix interpreter, allInstances is

implemented as a new primitive method.

Tektronix-Specific Primitives

The primitives specific to the Tektronix interpreter fall into seven categories:

- System and Display Calls – 4 primitives
  (Indexes 132, 133, 135, and 139)
- System Management – 4 primitives
  (Indexes 129, 130, 131, and 147)
- Object Management – 3 primitives
  (Indexes 137, 138, and 143)
- Instance Creation – 3 primitives
  (Indexes 140, 141, and 142)
- String Comparison – 1 primitive
  (Index 148)
- Floating Point – 6 primitives
  (Indexes 154, 155, 156, 157, 158, and 159)

System and Display Calls

These system call primitives support the OS interface:

- #132 – System call primitive
- #133 – Signal/semaphore primitive
- #135 – Display call primitive
- #139 – Access program parameters primitive

These primitives are not interruptable and have no garbage collection during execution of the primitive. The Smalltalk interpreter does not protect itself from invalid or incorrect system calls made with these primitives.

These primitives expect a receiver of the class UTekSystemCall. This object contains instance variables for registers d0, d1, d2, a0, and a1 (used as input registers to the system or display call, or as primitive parameters) and will return arguments for registers d0, d1, a0, and a1. The parameter is a field in the receiver.

System and Display Call Argument Translation. The display and operating systems have different representations of data than Smalltalk. While Smalltalk uses a unique object representation for data, the display system and operating system use a more general, processor-specific representation. The system call primitives must translate between these two representations.

The object types which may be passed as data to these include:

- smallintegers
- four byte large integers
- byte indexable objects
- nil
- instances of ExternalPointerData or its subclasses.

ExternalPointerData objects are identified by having two or more instance variables. The first is an instance of a byte indexable object which is used as a buffer area by the primitive. The second is a pointer indexable object which describes the data to be inserted into the buffer. The pointer indexable object consists of pairs of data and offsets: the data is any of the allowable data types above, while the offset is an integer one-based offset into the first instance variable, the byte indexable buffer object. The primitives scan the pointer object, translating the data and inserting the results into the buffer object.

The results of the translations for the buffer objects is a machine integer; for byte indexable objects, a pointer into the object's data area; for nil, a machine integer zero; for instances of ExternalPointerData, a pointer to the data area of its buffer object (which has the translated data from its pointer object).

System Call Primitive Failure. The system call primitives will fail if a data object passed is not one of the above types. These primitives will also fail if an offset in an ExternalPointerData pointer object falls outside the size of its associated buffer object, or if any byte indexable object is of zero length, or if the system call DOIn is not a valid system call number.

System and Display Call Primitives

Index	Class and Method Name	Functional Description
132	UTekSystemCall systemInvokeQuietly	This primitive is used to make all system calls (except for the special case of the signal/semaphor connection, which uses primitive 133). It will fail if the parameters are not SmallIntegers, 4-byte large integers, byte indexable objects, nil or instances of ExternalData or its subclasses. The return code parameter must be nil and the opcode must be a valid SmallInteger. Argument: None. Returns: false if the system call reported error. Otherwise, returns true.

Index	Class and Method Name	Functional Description
133	UTekSystemCall signalInvoke	This primitive is a system call which connects the receipt of a signal to a Smalltalk semaphore. It will fail if the parameters are not SmallIntegers, 4-byte large integers, byte indexable objects, nil or instances of Semaphore, ExternalData or its subclasses. The return code parameter must be nil and the opcode must be a valid SmallInteger. The parameters are: 0 – Connect Smalltalk semaphore or address. 1 – Ignore signal. 2 – Default action on signal receipt. The action-indicating parameter is passed in D0. When a semaphore is added to an interrupt, each time the interrupt is received the semaphore has a count added to its excess signals. Argument: None.
135	AimSystemCall displayInvoke	This primitive specifies a display-related system call to the operating system. It will fail if the parameters are not SmallIntegers, 4-byte large integers, byte indexable objects, nil or instances of ExternalData or its subclasses. The return code parameter must be nil and the opcode must be a valid SmallInteger. Argument: None. Returns: false if the system call reported error. Otherwise, returns true.
139	UTekSystemCall environmentInvoke	This primitive returns the address of the specified program parameter. This parameter is specified with the values given below. The parameters are: 1 – Command line arguments. 2 – Environment variables. 3 – Hardware/software configuration block. 4 – Interpreter version string. 5 – Copyright string. 6 – Operating System indicator. 7 – Time offset. Argument: None. Returns: false if the system call reported error. Otherwise, returns true. The return value of this primitive is in D0, and is a SmallInteger unless the requested value is not available, in which case the return value is nil. The return value of the operating system indicator for UTek is 2.

System Management Primitives

Index	Class and Method Name	Functional Description
129	SystemDictionary namedSnapshot:	This primitive causes a snapshot of the currently executing virtual image to be written to a file. It will fail if it can't write snapshot. Argument: instance of String. Returns: nil for successful write, self if the snapshot is reloaded, and false if it can't complete writing the snapshot file.
130	ContextPart primIncrementStackP	This primitive adds one to the stack pointer field of the receiving context and stores nil into the new top of stack element. This primitive and primitive #131 are the only acceptable ways to explicitly modify a context's stack pointer. Argument: None.
131	ContextPart primDecrementStackP	This primitive subtracts one from the stack pointer field of the receiving context. This primitive and primitive #130 are the only acceptable ways to explicitly modify a context's stack pointer. Argument: None.
147	SystemDictionary loadInterpreterKnownObjects	This primitive modifies the table of objects known to the interpreter and returns the difference in size. Argument: instance of Array. Returns: SmallInteger < 0 if array is smaller than interpreter array. SmallInteger = 0 if array is right size. SmallInteger > 0 if array is larger than interpreter array.

Object Management Primitives

Index	Class and Method Name	Functional Description
137	SystemDictionary garbageCollect:	This primitive forces a garbage collection. The argument identifies a storage grade. The virtual image is partioned into grades containing objects of corresponding ages. That is, newer objects are contained in lower grades and older objects are contained in upper grades. Valid numbers for grades are 0-7 inclusive. It will fail if the argument is out of range. Argument: integer 0 – 7. Returns: self
138	SystemDictionary core	Answer an Array containing the number of objects in the system and the number of words they occupy. Note that the count may include garbage objects which are eligible for garbage collection. Argument: None. Returns: an array with the first element = # of object, and the second element = # of words used (words are 32-bit quantities).
143	Behavior allInstances	Answer an array containing of all instances of this class. This may include instances that are elibible for garbage collection. Argument: None. Returns: an array of all objects having the class of the receiver.

Instance Creation Primitives

Index	Class and Method Name	Functional Description
140	DisplayBitmap basicNew: new:	Answer a new instance of DisplayBitmap with the number of indexable variables specified by the argument, anInteger. It will fail if the argument is not SmallInteger. Argument: SmallInteger Returns: instance of display bitmap.
141	ContextPart basicNew: new:	Answer a new instance of the receiver with the number of indexable variables specified by the argument, anInteger. Use of this instantiation primitive enables the creation of subclasses MethodContext and BlockContext. It will fail if the argument is not SmallInteger. Argument: SmallInteger Returns: instance of a Context.
142	Object shallowCopy	This primitive makes a shallow copy of the receiver. It can be applied to any object. Returns: a new instance which is a shallow copy of the receiver.

String Comparison Primitive

Index	Class and Method Name	Functional Description
148	String =	Answers true if the receiver and argument contain the same ASCII characters. Answers false if not. It will fail if the class of the argument is different from the class of the receiver. Returns: true if the strings are equal, false if otherwise.

Floating Point Primitives

Index	Class and Method Name	Functional Description
154	Float arcSin	Answers arcsine x, where x is the receiver. Returns: trig arc sin function (float).
155	Float arcCos	Answers arccosine x, where x is the receiver. Returns: trig arc cosine function (float).
156	Float arcTan	Answers arctangent x, where x is the receiver. Returns: trig arc tangent.
157	Float exp	Answers e^x, where x is the receiver. Returns: e^x function (float).
158	Float Ln	Answers ln x, where x is the receiver. Returns: log base e (float).
159	Float log	Answers log x, where x is the receiver. Returns: log base 10 (float).

Primitives with Different Functions

41 – 50 Float + through Float truncated – If the argument is a SmallInteger, it is converted to a Float number and there is no failure.

72 Object become: – This method is potentially "expensive" in the sense that in the Tektronix system it takes a relatively long time to execute. The primary reason for this is that the interpreters based on the Goldberg and Robson specification rely on swapping object table references – in comparison, the Tektronix interpreter must actually manipulate oops in memory (since there is no object table). Many special cases are optimized to minimize execution time but in the most general case, this primitive involves examining all objects in the virtual image.

In the Tektronix system, you may want to find alternative ways to code algorithms that use become:.

75 Object asOop, Object hash, Symbol hash – In the systems based on the Goldberg and Robson specification, an object's object table index (returned by asOop) is frequently used as a hash value. In the Tektronix system, each object is assigned a hash value at creation. This is a 16-bit value. asOop is defined to return this value. In the Tektronix system, asOop is not an invertible function, since there is no one-to-one correspondence between objects and asOop values. See the earlier discussion of asObject.

112 SystemDictionary coreLeft – This returns an *estimate* of the amount of memory available for new objects. (Use primitive 138 instead.)

115 SystemDictionary oopsLeft – This returns an *estimate* of the number of oops remaining to be allocated based on the core left value divided by the average object size. (Use primitive 138 instead.)

116 SystemDictionary signal:atOopsLeft:wordsLeft: – Since oops and memory are not practical system limits, this functions as a no-op.

Section 11
The Smalltalk Directories

OVERVIEW

This chapter describes the contents of the directories that you receive with the standard Smalltalk system.

Smalltalk is an interpreted language. The interpreter is an executable file called *smalltalk* residing in the */bin* directory. All other files in the standard Smalltalk system are shipped in the directory */usr/lib/smalltalk.*

THE DIRECTORIES

The */usr/lib/smalltalk* directory contains one file and five directories. They are:

- The *standardImage* file.
- The *system, fileIn, demo, textStyles,* and *conversion* directories.

The sections below each describe the contents of one of these items.

THE STANDARDIMAGE FILE

The *standardImage* file is the default Tektronix Smalltalk image. It is based on the Xerox Version 2 Smalltalk image and contains many Tektronix enhancements.

The system Directory

The */usr/lib/smalltalk/system* directory contains the following items:

- *standardSources.VersionTB2.2.1.* This file contains the standard source code for each of the methods in the standard image.
- *initialization.* This directory contains files used to initialize classes and perform other system support tasks. Users need not access these files.

The fileIn Directory

The */usr/lib/smalltalk/fileIn* directory contains Smalltalk code that may be added to an image. Included are useful class definitions, user interface enhancements and simple applications. The code in this directory is generally user-contributed and should be thought of as a starting point rather than a complete, debugged solution.

Each file contains comments about its function. File names ending in *.st* indicate that the file contains code to be filed in. File names ending in *.ws* create a workspace when filed in, containing code to be executed from a workspace.

Before filing in code, it is a good idea to open a ChangeListView and check to see if there are any conflicts with your system code before reading the file in. (See Section 6, **Change Management**, for a discussion of using ChangeListView.) After filing in new classes, update your browser by choosing the middle button update menu item.

The */smalltalk/fileIn* directory contains a variety of files. A *README* file is provided to fully document this directory. However, a few of the more useful and interesting files are described below.

- The files *findClass.st* and *extendedBrowser.st* both provide useful enhancements to the System Browser.
- The files *addTextStyleToSystemMenu.st* and *addTextStyleToYellowButtonMenu.st* both provide examples of adding text styles to otherwise standard system menus.
- The files *FinancialHistory.st* and *WireList-ASimpleMVCExample.st* both provide examples showing how the Model-View-Controller paradigm is used to create an application.
- The file *Example-Subtasking.st* provides an example of accessing the multitasking functions of the operating system from within Smalltalk.

NOTE

Code in the fileIn *directory is not supported. It is provided only as an aid.*

The demo Directory

The directory */usr/lib/smalltalk/demo* contains the following files and directory:

- *demoImage.* This is an image containing interesting visual applications.
- *demoChanges.* This is the changes file for *demoImage.*
- *makingADemoImage.st.* This file contains templates that are useful for reading in the forms, creating windows, and accessing other files used in *demoImage.*
- *forms.* This is a directory containing bitmaps. See the contents of *makingADemoImage.st* for workspace contents to create demos involving forms. These demos include static display of forms and primitive animation involving forms.
- A *README* file is also provided for more information.

Other files may also be found in the *demo* directory. Many files whose names end with the suffix *.st* provide the source code for games which can be played in Smalltalk.

NOTE

Code in the demo *directory is not supported. It is provided for demonstration purposes only. Users are welcome to create their own versions of the demo image.*

The textStyles Directory

The directory */usr/lib/smalltalk/textStyles* contains a variety of files. Each file contains a collection of related expressions used to create new text styles within Smalltalk. These expressions must be selected and evaluated (using the doit menu item in a fileList or workspace) to create a new text style. Experiment by filing in these files to see the effect different text styles have on your Smalltalk image. (See Section 7, **Fonts in Smalltalk**, for a more complete discussion of text styles.)

The conversion Directory

The */usr/lib/smalltalk/conversion* directory contains Smalltalk source code used to update older versions of UTek Smalltalk images to the currently shipped version.

NOTE

For Smalltalk Version TB2.2.1, this directory is empty.

Appendix A
Smalltalk Books Information

ERRORS IN THE ADDISON-WESLEY BOOKS

The following is a list of errors in the Goldberg book (imprint 1984):

- On page 83, in Figure 5.5, the syntax diagram for "symbol" needs a path out analogous to "keyword".
- On page 88, in the first paragraph, ((sum count)) should be ((sum/count)).
- On page 126, the icons for saving and retrieving a FormEditor form source should be reversed.

The following is a list of the errors in the Goldberg and Robson book, which was reprinted with corrections, May 1983.

- On pages 127 and 129, it should refer to printStringRadix: instead of radix:.
- On page 146, in Figure 10.1 the cartouche (rounded box) that holds the question ''accessible by a key?'' should read something like ''fixed-size?''
- On page 161, LinkedList should be Link. This replacement should be made in the title to the instance protocol table at the bottom of the page.
- On page 202, the WriteStream instance protocol should list crtab and crtab: instead of crTab and crTab:.
- On page 289, it says to add the method to the instance creation protocol of class Class, but it should be added to the instance protocol named subclass creation.
- On pages 333 and 338, it refers to the class Bitmap. This class has been renamed WordArray.
- On page 399, it shows the wait cursor as three dots instead of the current hourglass shape. The message to Cursor for the crosshair is crossHair instead of crosshair.

Appendix B
Smalltalk Internal Character Codes

The *Keyboard and Mouse Functions* section of the *4405/06 Option 15 UTek Exceptions and Extensions* manual contains a table of Event Manager Key Codes. These raw key codes are mapped by the InputSensor class to an internal representation. This means that ''shifted s'' has an internal value that is different from an ''unshifted s'', which is also different from a ''control s'' and a ''control shifted s''. The internal values are the ones used by the ParagraphEditor.

Alphanumeric Keys

Table B-1 shows the Smalltalk Internal Character Code meanings for the main part of the keyboard — the ''alphanumeric keys.''

In this table, control characters are represented by the standard two- or three-letter abbreviations, given in ANSI X3.4 and ISO 646. Special symbols are represented by the four-character codes assigned to those symbols in ISO 6937. These meanings of these four-character codes are given in nearby notes.

Table B-1
Alphanumeric Keys

Smalltalk Internal Character Codes Standard North American Keyboard															
Row 1 Keys	{	!	@	#	$	%	^	&	*	(	)	SP09	+	}	
(Mode)	[	1	2	3	4	5	6	7	8	9	0	SP10	=	]	RUB
Unshifted	91	49	50	51	52	53	54	55	56	57	48	45	61	93	127
Shifted	123	33	64	35	36	37	94	38	42	40	41	95	43	125	127
Ctrl	132	136	144	143	128	130	129	131	180	149	135	137	6	29	127
Ctrl-Shifted	249	223	208	207	192	191	30	195	244	213	0	31	14	29	127
Row 2 Keys		~											`		
(Mode)	ESC	|	Q	W	E	R	T	Y	U	I	O	P	\	BSP	LF
Unshifted	27	124	113	119	101	114	116	121	117	105	111	112	92	8	10
Shifted	27	126	81	87	69	82	84	89	85	73	79	80	96	8	10
Ctrl	27	133	17	23	5	18	20	25	21	150	15	16	28	8	10
Ctrl-Shifted	27	134	203	209	194	239	240	242	197	214	216	202	28	8	10
Row 3 Keys											:	"			
(Mode)	TAB	A	S	D	F	G	H	J	K	L	;	'	RTN		
Unshifted	9	97	115	100	102	103	104	106	107	108	59	39	13		
Shifted	9	65	83	68	70	71	72	74	75	76	58	34	13		
Ctrl	9	1	19	4	6	7	8	10	11	12	3	138	13		
Ctrl-Shifted	9	212	211	196	226	241	243	229	200	217	3	219	13		
Row 4 Keys								<	>	?					
(Mode)	Z	X	C	V	B	N	M	,	.	/					
Unshifted	122	120	99	118	98	110	109	44	46	47					
Shifted	90	88	67	86	66	78	77	60	62	63					
Ctrl	26	24	3	22	2	14	13	1	18	27					
Ctrl-Shifted	231	215	228	198	230	245	246	218	233	203					
Row 5 Keys															
(Mode)	SPC														
Unshifted	32														
Shifted	32														
Ctrl	32														
Ctrl-Shifted	32														
Notes:	SP09: "low line" or underline SP10: hyphen or minus sign														

Numeric Pad Keys

The numeric pad is located to the right of the main set of alphanumeric keys.

Table B-2
Numeric Pad Keys

Smalltalk Internal Character Codes Standard North American Keyboard														
Key Pad Name Mode	0	1	2	3	4	5	6	7	8	9	.	,	-	ENT
Unshifted	48	49	50	51	52	53	54	55	56	57	46	44	45	27
Shifted	181	182	183	184	185	186	187	188	189	190	175	177	176	178
Ctrl	255	255	255	255	255	255	255	255	255	255	255	255	255	255
Ctrl-Shifted	255	255	255	255	255	255	255	255	255	255	255	255	255	255

Joydisk Keys

The joydisk is located to the upper left of the main set of alphanumeric keys.

Table B-3
Joydisk Keys

Smalltalk Internal Character Codes Standard North American Keyboard				
Joydisk Key Name Mode	Up	Down	Right	Left
Unshifted	255	255	255	255
Shifted	255	255	255	255
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255

Function Keys

The function keys F1-F12 are grouped in three groups of four keys and are located in a row above both the alphanumeric keys and the numeric key pad.

Table B-4
Function Keys

Smalltalk Internal Character Codes Standard North American Keyboard				
Function Key Name Mode	F1	F2	F3	F4
Unshifted	151	152	153	154
Shifted	163	164	165	166
Ctrl	145	146	147	255
Ctrl-Shifted	255	255	255	255
Function Key Name Mode	F5	F6	F7	F8
Unshifted	155	156	157	158
Shifted	167	168	169	170
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255
Function Key Name Mode	F9	F10	F11	F12
Unshifted	159	160	161	162
Shifted	171	172	173	174
Ctrl	255	255	255	255
Ctrl-Shifted	255	255	255	255

Special Function Keys

There are only two special function keys on the Pegasus keyboard. One is the "up-arrow/left-arrow" key in the upper left corner of the main key area, while the other is the BREAK key in the lower right corner of the main key area.

Table B-5
Special Function Keys

Default ANSI Meanings of Special Function Keys Standard North American Keyboard		
Function Key Name Mode	↑ ←	BREAK
Unshifted	94	179
Shifted	95	179
Ctrl	30	179
Ctrl-Shifted	148	179

Appendix C
Smalltalk Printing Facilities

PRINTING SMALLTALK BITMAP FILES

Look in the */usr/samples/printer* directory for a C program, *bprint.c*, that prints Smalltalk forms or bitmaps on a Tektronix 4644 printer. You can either use this program as it stands if you have the 4644 printer or you can modify the program to be compatible with a different printer.

This program, *bprint.c*, prints Smalltalk bitmaps as generated by the screenCopy menu item or from a fileOut of a specific form. If you modify the program, the default graphic density and screen width pixels per printer line should be determined by the characteristics of your printer. In *bprint.c*, the default graphic density is double. Option "+s" enables single-density mode which gives you a larger image but with possible truncation.

Printer Support

The copy display command in the System Menu prompts for a file name and then copies the screen bitmap to that file. This file is written in the same format as that of a Form. The file can then be sent to the printer from the operating system. To print a bitmap from a file called screen.bm on a Tektronix 4644 printer, type:

/usr/samples/printer/bprint screen.bm

The *bprint* command defaults to double density, which means that the printout looks half as wide as it should in relation to its height. If you use the *+s* option on the *bprint* command, the aspect ratio looks much more like the real screen, but the last few pixels on the right side of the screen are missing.

The print out menu option in the System Browser writes the code out to a file with a *.pp* extension. This code cannot be directly filed into a Smalltalk-80 image, but is intended to provide a more human readable format than that provided by the file out menu selection. It can be regarded as a pretty printed version, although no automatic formatting is performed on methods.

The *.pp* files contain control characters for bold, italic, and normal fonts taken from from ANSI Standard X3.64.

From the operating system, you can look at a file named *Collection.pp* by typing:

list Collection.pp

(There are no italic fonts for the terminal, so italic entries are printed in the normal font.)

To print the file on a Tektronix 4644 printer, type:

/samples/printer/print +s Collection.pp

An example of this output might look like:

```
Object subclass: #Collection
        instanceVariableNames: ´´
        classVariableNames: ´´
        poolDictionaries: ´´
        category: ´Collections-Abstract´

Collection comment: ´I am the abstract class of all collection classes.´

Collection methodsFor: ´accessing´
```

size

```
        "Answer how many elements the receiver contains."

        | tally |
        tally ← 0.
        self do: [:each | tally ← tally + 1].
        ↑tally


Collection methodsFor: ´testing´
```

includes: anObject

```
        "Answer whether anObject is one of the receiver´s elements."

        self do: [:each | anObject = each ifTrue: [↑true]].
        ↑false
```

isEmpty

```
        "Answer whether the receiver contains any elements."

        ↑self size = 0
```

occurrencesOf: anObject

```
        "Answer how many of the receiver´s elements are equal to anObject."

        | tally |
        tally ← 0.
        self do: [:each | anObject = each ifTrue: [tally ← tally + 1]].
        ↑tally
```

Index