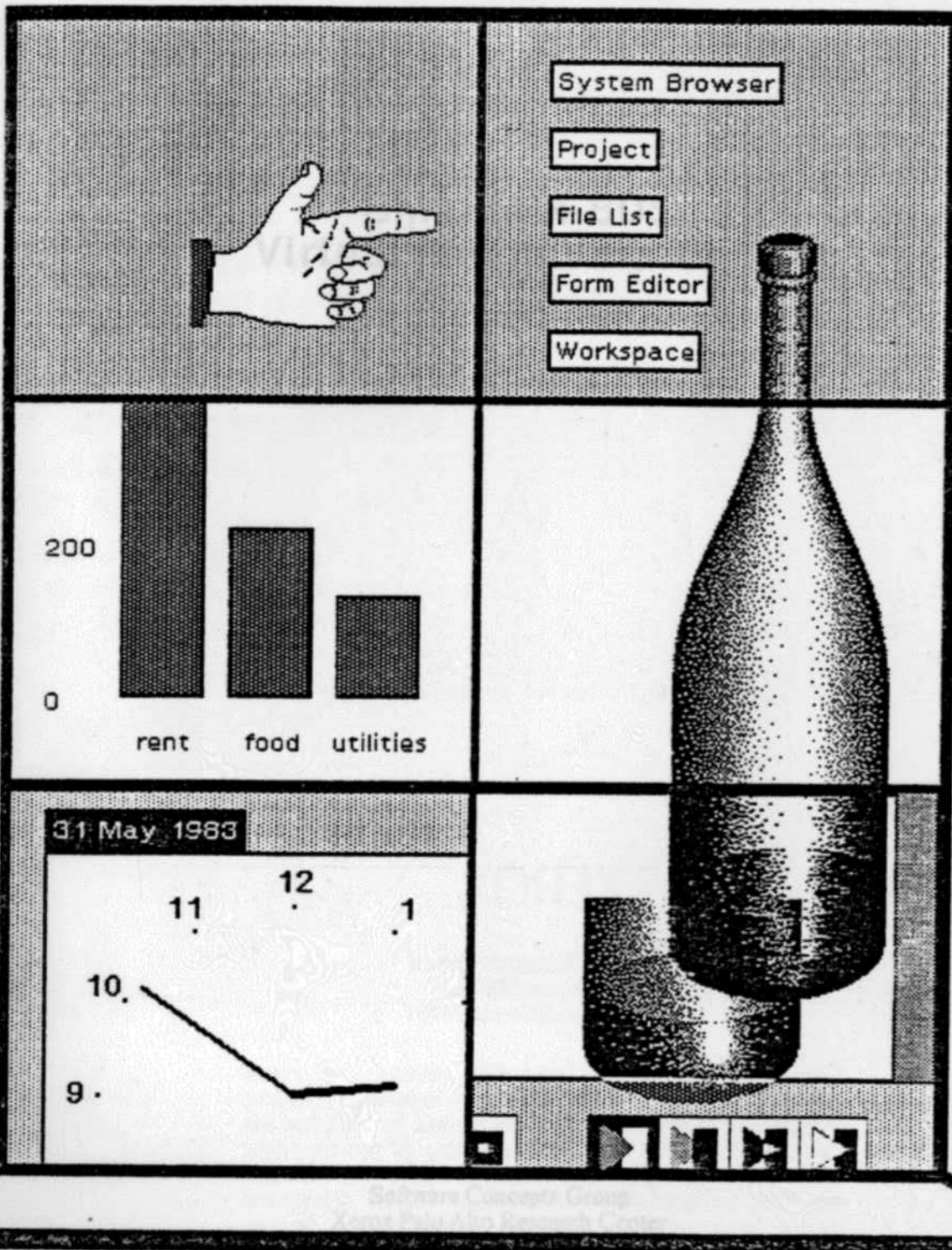


# SMALLTALK-80

## Virtual Image Version 2



**System Concepts Laboratory**  
**Xerox Palo Alto Research Center**

# **Smalltalk-80: Virtual Image Version 2**

**Software Concepts Group  
Xerox Palo Alto Research Center**

# XEROX

Palo Alto Research Center  
3333 Coyote Hill Rd.  
Palo Alto, California 94304

Copyright © 1983 by Xerox Corporation. All rights reserved. The Virtual Image described and/or reproduced in this document was created in 1981 but has not been published within the meaning of the copyright law. The Virtual Image is furnished under a license and may not be used, copied, disclosed and/or distributed except in accordance with the terms of said license.

Smalltalk-80 is a Trademark of Xerox Corporation

# CONTENTS

**Part 1: Tape Format Description**

**Part 2: The Goodie Files**

**Part 3: Multiple Inheritance Description**

**Part 4: Errata in *Smalltalk-80: The Language and its Implementation***

**Part 5: Summary of System Interface Components**



# Part 1: Tape Format Description

## Scope

*Smalltalk-80: The Language and its Implementation* (Adele Goldberg and David Robson, Addison-Wesley, Reading, Massachusetts, 1983) contains information necessary to implement the Smalltalk-80 virtual machine, including the list of bytecodes and their implementations, the list of primitives and their implementations, and a description of storage management. The book, however, does not contain specifics for any given Smalltalk-80 virtual image, as delivered on a magnetic tape. This memo is intended to supply those specifics.

This document assumes that the reader has read Part 4 of the book, or at least has an understanding of the terminology used in that part, such as the terms 'oop' and 'object table.'

Included below is a description of the file formats for the accompanying tape, which contains the Smalltalk-80 Virtual Image Version 2, and a list of those objects and classes known to Smalltalk-80 interpreters.

## What is on the Tape

The Smalltalk-80 Virtual Image and associated files are written on a 9-track, 1600 bpi phase-encoded magnetic tape. The tape consists of binary files written in 'continuous stream' mode, with 2048 byte records, and an eof mark after each file. The files (in order of appearance on the tape) are:

1.	Virtual Image	292 records	596,128 bytes
2.	Sources file (in the image, called Smalltalk80.sources)	689 records	1,409,645 bytes
3.	Changes file (in the image, called Smalltalk80.changes)	1 record	93 bytes
4.	Trace file 1 of simulator	17 records	32,897 bytes
5.	Trace file 2 of simulator	11 records	20,644 bytes
6.	Trace file 3 of simulator	12 records	23,679 bytes
7.	List of object pointers for classes	7 records	14,171 bytes
8.	List of object pointers for methods	94 records	192,245 bytes
9-17.	Goodie files (described in the following sections)		

Note that the last record of each tape is a full, 2048-byte record, padded with nulls (0's) from the end of the file to the end of the record. The byte lengths above are the size of the actual data part of the file, and do *not* include the padding. The padding bytes must be stripped for the goodie files; whether they should be stripped for the other files depends on your system—it would probably be best to strip them all.

## System Dependencies

All bytes are considered 8-bits, all words are 16-bits. Words in the file are stored in the order of more significant byte followed by less significant byte.

We realize that some implementations would prefer to have words stored low byte followed by high byte. Unfortunately, there is not one consistent "other" way to store bytes in words. We think that the transformation of the image that would work for most machines is to swap the bytes of all fields accessed as words, and to not swap the fields accessed as bytes. This has been done by a number of other implementations already (see *Smalltalk-80, Bits of History, Words of Advice*, Glenn Krasner, Ed., Addison-Wesley Publishing Co., Reading Massachusetts, September, 1983). The transformation typically works best by having an auxiliary program translate the image from its *interchange format* to your *internal format*. The suggested algorithm which may or may not be best for your system is

For word-type objects: swap every field.

For CompiledMethods: swap Length, Class, Header and Literal fields only.

For all other byte-type objects: swap Length and Class fields only.

## Virtual Image File (File 1)

The Virtual Memory Image consists of length information, followed by the data representing objects (object space), followed by the data representing the object table. The first four bytes (stored as most significant byte first) contain the length of the object space (in 16-bit words). The next four bytes of the file is the length of the object table. The next 504 bytes are set to 0. By convention, an image file is defined to be in interchange format if the ninth and tenth bytes are zero—if either byte is non-zero the image file is assumed to be in an internal, non-interchange format.

For this image, the first ten bytes are:

0, 3, 363<sub>8</sub>, 100<sub>8</sub>, (Object space length = 517760<sub>10</sub>bytes)

0, 0, 227<sub>8</sub>, 120<sub>8</sub>, (Object table length = 77472<sub>10</sub>bytes)

0, 0

(and the last ten bytes of the file are:

0, 0, 0, 40<sub>8</sub>, 0, 0, 1, 103<sub>8</sub>, 363<sub>8</sub>, 73<sub>8</sub>).

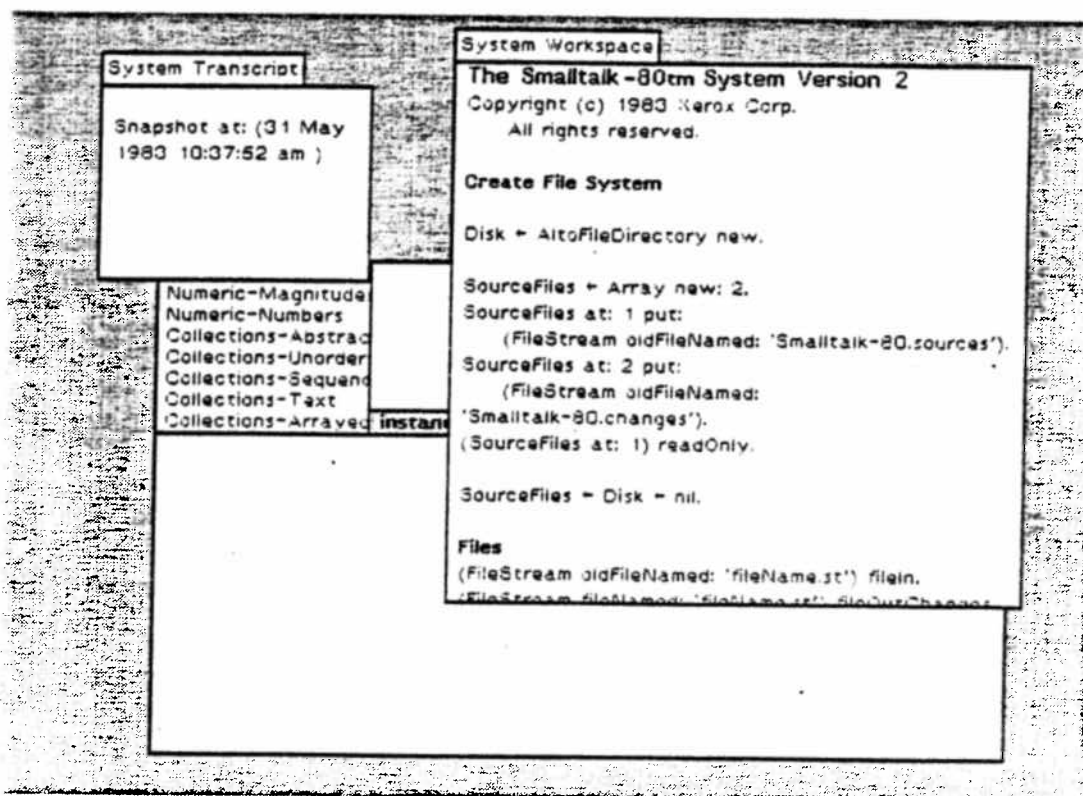
Following this (starting at the 513th byte) is the object space. The first word encountered is the first word of the object whose object pointer (oop) is 2 and whose object space address is 0 (20 bit address). (Oop 0 is reserved as an invalid oop; oop 1 and all the other odd oops are SmallIntegers.) The next words are the fields of the remaining objects, stored consecutively, up to the length of the object space. Following the object space are enough 0's to start the object table at a page (256-word, 512-byte) boundary.

The next word is the first word of the object table entry for the object with oop (object pointer) = 0. (Oop 0 is an invalid oop, but the object table entry exists anyway.) The rest of the words of the object table follow. The last word of the object table is the last word of the file (once the padding bytes mentioned above are stripped off).

**Note:**

- The length and object space portions of the file are padded with 0's to the end of a page, but the object table is not.
- The object table entries either are pointers to objects in the object space or are unused entries, and the object space contains only objects; there is no free memory in the object space.
- The object table contains unused entries. These have the 'freeEntry' bit set, but all other bits in both words are 0. Implementors who want to link the unused entries, as described in the book, will have to link these themselves.
- The object table assumes that objects are stored contiguously starting at address 0 in a 20-bit address space. There is no distinction for "segment" boundaries, as described in the book. Any desired address translation is left to the implementors.

When the system comes up, there will be three views alive on the screen (see figure), a SystemTranscript, a SystemWorkspace, and a SystemBrowser. The screen will be 640 by 480 pixels in size (there is an expression in the SystemWorkspace for changing it). The globals Disk and SourceFiles will both be set to nil so that source files will not be accessed, but instead methods will be decompiled. The selected text in the SystemWorkspace is what to execute to build an Alto file system and to initialize SourceFiles. Once your file system is working, you can use this as a guide to installing it.



### Sources and Changes Files (Files 2 and 3)

The second file of the tape is a copy of the system sources, which you may print if you like. This file consists of the definition of each class in the system, followed by the code for that class's methods. (There is a form feed character between each class, ASCII 12<sub>10</sub>.) The format of this file may be understood by reading the source code for the nextChunk method in class ReadWriteStream and the getSource method in class CompiledMethod; it is the same format as that used in fileIn/fileOut.

Each CompiledMethod in the image contains a pointer to its source code, encoded in the last three bytes of that method. The two msb's of the first of these bytes determine the file on which the source is stored (00=Smalltalk-80.sources', 01=Smalltalk-80.changes', 10=unused, 11=unused). The six lsb's of this byte with the two following bytes make up a 22-bit pointer specifying where in that file the source code begins. The source code for that method is terminated by \$! (any embedded \$! is doubled).

#### Note:

The sources file on this tape is quite long; and the changes file is very short.

### Trace Files (Files 4-6)

The tape includes three traces of the Smalltalk-80 interpreter executing the first bytecodes in this Smalltalk-80 virtual image. These were made by running the formal specification of the interpreter written in Smalltalk-80 itself. The intention is that you would follow these traces, by hand for the first while and later by writing your own trace files, and compare them with your system's performance—where they differ you are likely to have an implementation bug. The three traces show decreasing levels of detail over increasing durations.

- The first trace shows all memory references, allocations, bytecodes, message transmissions, returns and primitive invocations for about the first 100 bytecodes executed.
- The second trace shows only the bytecodes, message transmissions, returns and primitives for about the first 512 bytecodes.
- The third trace shows message transmissions, primitives and returns for about the first 1982 bytecodes. The lines of this trace are indented according to the level of method invocation (i.e., the depth of the context stack).

The format of each type of entry is given below. All numbers are shown in decimal.

#### Memory Reference (only in first trace)

**Pointer Fetch**

*object-pointer* pointer: *fieldIndex* = *field contents*  
 E.g., 20656 pointer: 20 = 1617

**Byte Fetch**

*object-pointer* byte: *byteIndex* = *byte contents*  
 E.g., 3872 byte: 46 = 208

**Word Fetch**

*object-pointer* word: *fieldIndex* = *field contents*  
 E.g., 18168 word: 0 = 5

**Pointer Store**

*object-pointer* pointer: *fieldIndex* ← *field contents*  
 E.g., 20654 pointer: 1 ← 15

**Allocation**

allocating oop: *object-pointer*  
 E.g., allocating oop: 20654

**Bytecodes (in first and second traces)**

Bytecode <*bytecode-index*> *bytecode-description*  
 E.g., Bytecode <16> Push Temporary Variable 0

**Message Transmission (in all traces)**

[*cycle=bytecode cycle*] *receiver-description selector-string argument-descriptions*  
 The bytecode cycle is the number of bytecodes that have been executed. The receiver and argument descriptions will show the class of the appropriate object except in the case of SmallIntegers, Strings, true, false, and nil, which print more nicely.

E.g.,           [cycle = 408] aLargePositiveInteger digitAt: 3  
                  [cycle = 75] 40 digitMultiply:neg: 808 false

**Primitive Invocations (in all traces)**

Primitive #*primitive-index*  
 E.g., Primitive #70

**Returns (in all traces)**

↑ (method / block) of *returned value description*  
 E.g.,           ↑ (method) of aLargePositiveInteger  
                  ↑ (block) of 64

**Object Pointers List Files (Files 7 and 8)**

As an aid to debugging, the tape includes a list of the oops of all classes and methods in the system. These lists have been found particularly helpful in debugging the message lookup code of an interpreter. The list shows the oops in both octal and hexadecimal. Examples of classes from file 7 are

8r14	16rC	SmallInteger
8r16	16rE	String
8r20	16r10	Array
8r24	16r14	Float
8r26	16r16	MethodContext
8r30	16r18	BlockContext
8r32	16r1A	Point
8r34	16r1C	LargePositiveInteger
8r36	16r1E	DisplayBitmap
8r40	16r20	Message

which means that in the image, for example, class SmallInteger has oop  $14_8$  ( $0C_{16}$ ) and class Message has oop  $40_8$  ( $20_{16}$ ). Examples of methods from file 8 are

8r144	16r64	<Collection class>with:with:with:with:
8r150	16r68	<Collection class>with:
8r152	16r6A	<Collection class>with:with:with:
8r154	16r6C	<Collection class>with:with:
8r176	16r7E	<ClassOrganizer class>new
8r472	16r13A	<Object>nextInstance
8r500	16r140	<Object>printString
8r524	16r154	<Stream class>new
8r572	16r17A	<Stream>nextPut:
8r574	16r17C	<Stream>contents

which means that in the image, for example, the method for the message with:with:with:with: in class Ciollection class has oop  $144_8$  ( $64_{16}$ ) and the method for the message contents in class Stream has oop  $574_8$  ( $17C_{16}$ ).

## Objects Known to Interpreters

There is a set of objects that must be known by a Smalltalk-80 interpreter. For example, in various places in an interpreter, the oop of nil must be known to be 2. Typically, an interpreter will keep the oops of these objects as compile-time or run-time constants, or keep them in special tables. A list of these special oops/objects follows. Those marked \* are not necessarily needed by interpreters, but are included in this table as debugging aids.

2		the object nil
4		the object false
6		the object true
10 <sub>8</sub>	08 <sub>16</sub>	an Association whose value field is Processor
*12 <sub>8</sub>	0A <sub>16</sub>	Symbol class variable USTable, the table of Symbols
14 <sub>8</sub>	0C <sub>16</sub>	class SmallInteger
16 <sub>8</sub>	0E <sub>16</sub>	class String
20 <sub>8</sub>	10 <sub>16</sub>	class Array
*22 <sub>8</sub>	12 <sub>16</sub>	an Association whose value field is the SystemDictionary, Smalltalk
24 <sub>8</sub>	14 <sub>16</sub>	class Float
26 <sub>8</sub>	16 <sub>16</sub>	class MethodContext
30 <sub>8</sub>	18 <sub>16</sub>	class BlockContext
32 <sub>8</sub>	1A <sub>16</sub>	class Point
34 <sub>8</sub>	1C <sub>16</sub>	class LargePositiveInteger
*36 <sub>8</sub>	1E <sub>16</sub>	class DisplayBitmap
40 <sub>8</sub>	20 <sub>16</sub>	class Message
42 <sub>8</sub>	22 <sub>16</sub>	class CompiledMethod
*44 <sub>8</sub>	24 <sub>16</sub>	symbol # unusedOop18
46 <sub>8</sub>	26 <sub>16</sub>	class Semaphore
50 <sub>8</sub>	28 <sub>16</sub>	class Character
52 <sub>8</sub>	2A <sub>16</sub>	symbol # doesNotUnderstand:
54 <sub>8</sub>	2C <sub>16</sub>	symbol # cannotReturn:
*56 <sub>8</sub>	2E <sub>16</sub>	symbol # monitor:
60 <sub>8</sub>	30 <sub>16</sub>	SystemDictionary class variable SpecialSelectors, the array of selectors for bytecodes 260 <sub>8</sub> -317 <sub>8</sub>
62 <sub>8</sub>	32 <sub>16</sub>	Character class variable CharacterTable, table of Characters
64 <sub>8</sub>	34 <sub>16</sub>	symbol # mustBeBoolean





## Part 2: The Goodie Files

In addition to the Virtual Image, source code files, and system traces, we have also included on the tape nine files that make up eight *goodies*. The goodies are fairly small Smalltalk-80 applications. Some of the goodies are application examples from the books, others are new and have not been described elsewhere.

These goodies should serve as examples to new Smalltalk-80 programmers. However, they should *not* be considered complete, debugged software, but rather starting points for more complete and better debugged applications.

A second use of these goodies is to open the channels of communication among Smalltalk-80 users. It is our hope that a community of Smalltalk-80 programmers will develop, to encourage and facilitate the sharing of code. These goodies serve as the first pieces of code to share. In fact, two of them did not originate at Xerox PARC; one was written by a Smalltalk-80 programmer at Tektronix and the idea for another came from a Smalltalk-80 programmer at U.C. Berkeley. Please share any new goodies you have, including improvements to these, with the rest of us. The address to use in submitting goodies is:

Smalltalk-80 Newsletter Coordinator  
Xerox Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto CA 94304

The following pages describe the goodies in more detail. The format used in the descriptions is described in Part 5. Here we will list the files as they appear on the tape (with record counts as a double-check of the integrity of the tape). The goodie files are:

9.	Book Index Browser	11 records	22,187 bytes
10.	Book 1 Index	17 records	34,721 bytes
11.	Protocol Browser	2 records	3,117 bytes
12.	Project Browser	2 records	3,420 bytes
13.	Financial History	4 records	7,021 bytes
14.	Clock	3 records	4,757 bytes
15.	Animation	4 records	7,465 bytes
16.	Pointing Hand	3 records	5,846 bytes
17.	"Toothpaste" Graphics	1 record	1,339 bytes.

**An Index List**

\*

BankTeller  
 become:  
 Behavior  
 Bernoulli  
 binary message  
 BinaryTree  
 Binomial  
 Birtwistle, Graham  
 bit  
     fields  
     manipulation  
 BitBit  
 BitBit  
     combination rule  
 BitBitSimulation  
 Bitmap  
 bitmap  
 block  
     argument  
     context

**BlockContext**

Boolean  
 bordering

Key: BlockContext

Pages: 250, (254-255), 462, 559-560, 580-581, 583-585,  
619, 637-639

See:

See also: block (subject index)

Category: classes

## 1. Book Index Browser (Files 9 and 10)

This "goodie" is a collection of Smalltalk-80 class definitions that allow one to browse and modify the index for a book. This browser was used to create the indexes for the Smalltalk-80 book series. File 9 contains the class definitions, and file 10 is the actual index database for the book, *Smalltalk-80: The Language and its Implementation*.

<b>name</b>	<b>Book Index Browser</b>
<b>general description</b>	Used to browse and modify an index for a book. It includes support for adding and removing definition and use pages, for nesting entries under others, for "see" and "see also," for automatically creating range notation (2-4) out of sequential entries (2,3,4), and for filtering the viewed entries.
<b>how accessed</b>	<p>To bring the class definitions into your system, file in File 9 from the tape. Also bring File 10 from the tape into your file system, and call it, for example, "book1.index." The classes on this file are all put into system category <b>Book-Index</b>, so you may need to update your browser by choosing hellow button menu item update. To create a view, evaluate the comment in <b>IndexCardCollection openOn:</b> which is</p> <pre>IndexCardCollection openOn: 'book1.index'</pre> <p>The view created has three subviews arranged vertically. The top subview (subview A) is used to define the filter for the entries to be seen. The middle subview (subview B) is used to list, select, and modify index entries. The bottom subview (subview C) is a read-only view of the currently selected entry.</p>
<b>how created</b>	Evaluate the code above, then designate the area for the view with red button.
<b>how terminated</b>	Blue button choose close.
<b>blue button activity</b>	Default
<b>red button activity</b>	
subview A	Behaves like a standard paragraph editor.
subview B	Selects (or deselects) the entry to which the cursor points. When one is selected, the bottom subview will display its contents.

## An Index List

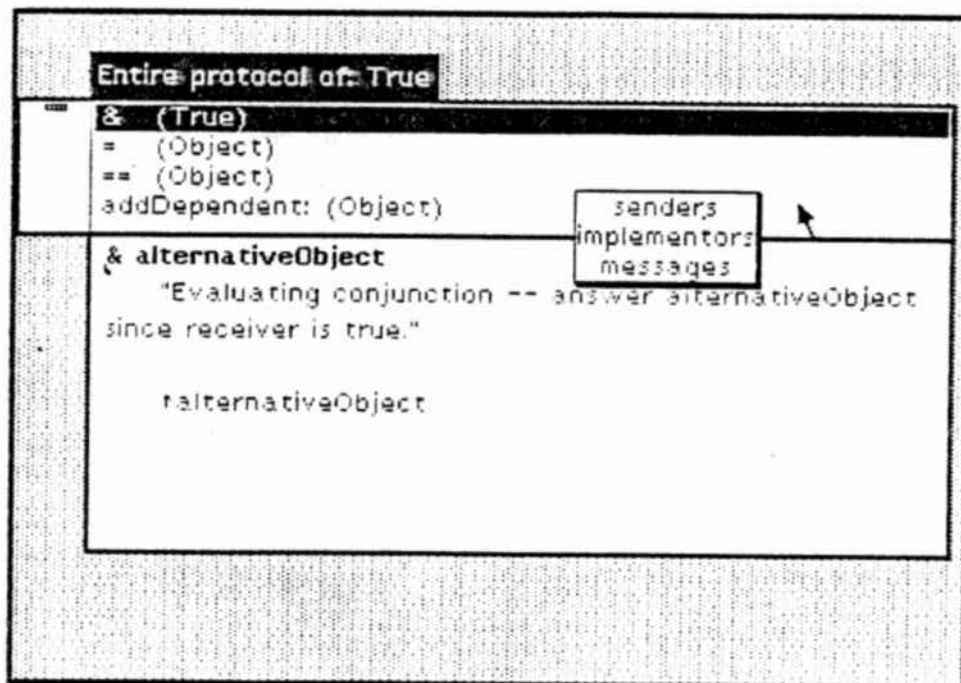
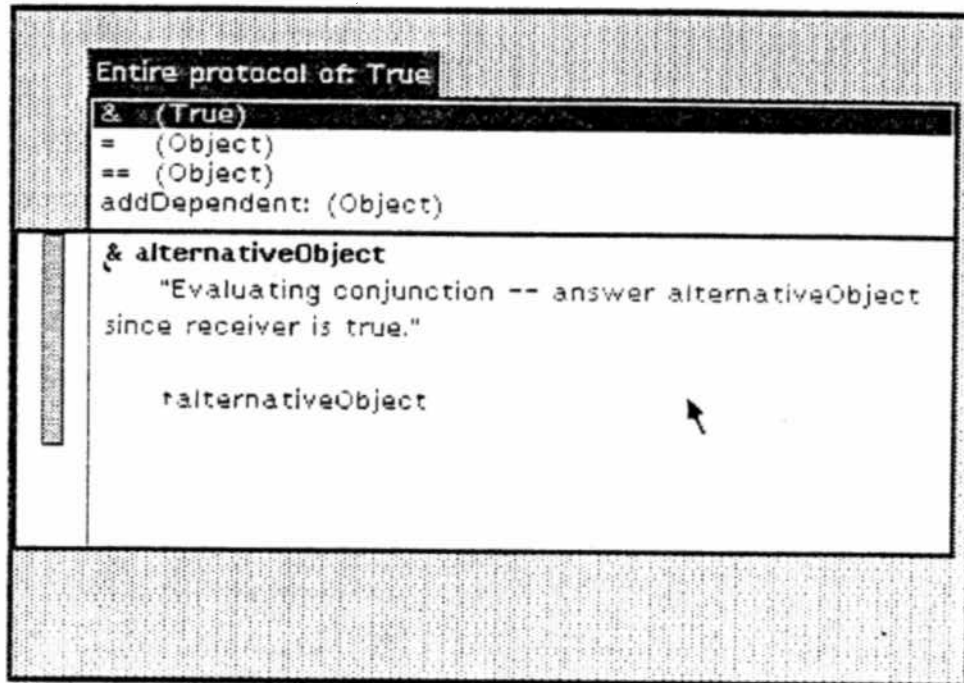
* ▲	
BankTeller	
become:	
Behavior	
Bernoulli	
binary message	
BinaryTree	add page
Binomial	add defn page
Birtwistle, Graham	add use page
bit	remove page
fields	new card
manipulation	new nested card
BitBit	remove card
BitBit	change key
combination rule	change nesting
BitBitSimulation	change category
Bitmap	change See
bitmap	change See Also
block	update file
argument	make ranges
context	
<b>BlockContext</b>	
Boolean	
bordering	
Key: BlockContext	
Pages: 253, (254-255), 462, 559-560, 580-581, 583-585, 619, 637-639	
See:	
See also: block (subject index)	
Category: classes	

## yellow button activity

subview A	accept	Accept the contents of the view to be the filter of the entries. Valid filters are: * Show every entry #<number> Show only entries on page <number>. <string> Show only entries whose category matches <string>
subview B	add page	Add a new page or page range to the selected entry. When prompted, you specify a page by the page number (followed by accept) or a page range by the start and end page numbers of the range with a minus (\$-) between them (followed by accept).
	add defn page	Add a new page or page range to the selected entry for its definition. Definition pages are identified in the bottom subview by being surrounded by parentheses.
	add use page	Add a new page or page range to the selected entry for its use. Use pages are identified in the bottom subview by being surrounded by angle brackets.
	new card	Add a new entry to the list. You will be prompted for the new key and category.
	new nested card	Add a new entry nested under the selected entry.
	remove card	Remove the selected entry.
	change key	Change the key of the selected entry.
	change nesting	Change the nesting of the selected entry.
	change category	Change the category of the selected entry.
	change See	Change the "See" reference of the selected entry.
	change See Also	Change the "See also" reference of the selected entry.
	update file	Store the current version of the index on a file (note although you read the index from a file, the file is not updated unless you give this command).
	make ranges	Run through all the entries, condensing sequential pages into page ranges.

## keyboard activity

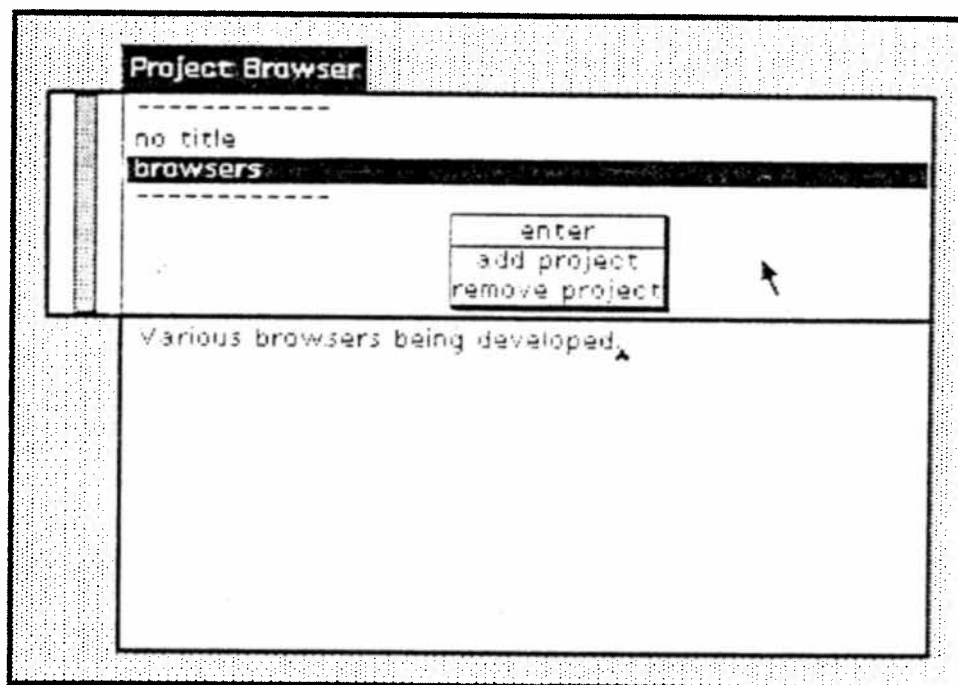
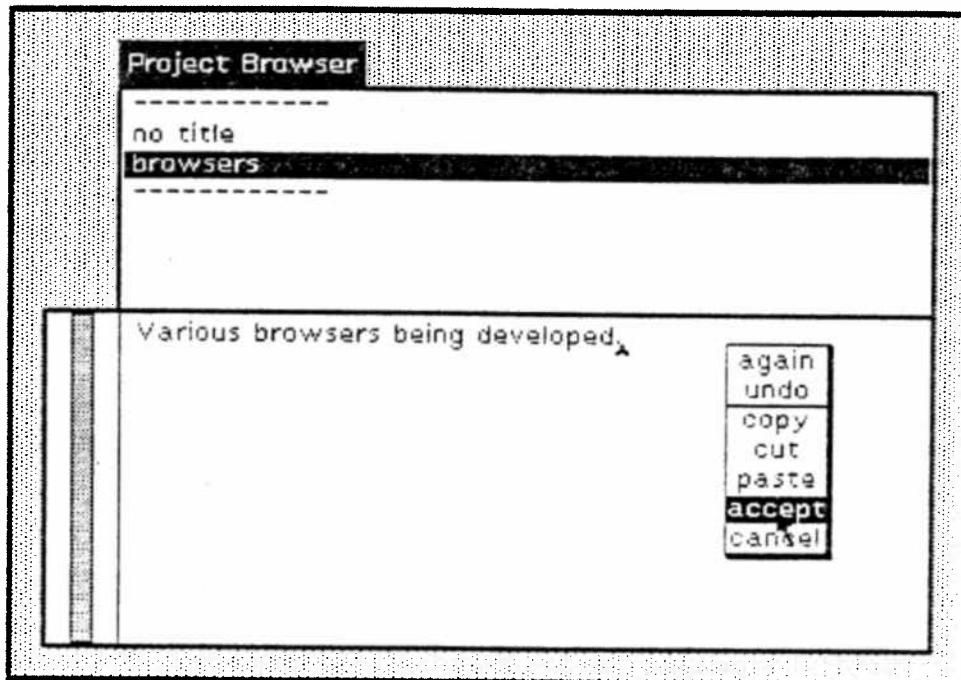
Subview A acts like a paragraph editor, the other subviews ignore keyboard activity.



## 2. Protocol Browser (File 11)

This "goodie" creates a browser to look at the entire protocol of a class, that is, all the messages an instance of that class can receive. It includes the messages defined in that class and all of its superclasses. (Since this is supposed to be the "external" protocol of a class, methods overwritten in a subclass are not included in the list, even though they can be accessed "internally" with `super`.) File 11 contains the code for creating a protocol browser as described in the book, *Smalltalk-80: The Interactive Programming Environment* (Adele Goldberg, Addison-Wesley Publishing Company, Reading, Massachusetts, November, 1983).

<b>name</b>	Protocol Browser
<b>general description</b>	Used to create a browser to view the entire protocol of a class.
<b>how accessed</b>	<p>To bring the class definitions into your system, file in File 11 from the tape. The class on this file is categorized under <code>Interface-Protocol</code>, so you may need to update your browser by choosing yellow button menu item <code>update</code>. To create a view, evaluate the comment in <code>ProtocolBrowser openForClass:</code> which is</p> <pre>ProtocolBrowser openForClass: ProtocolBrowser</pre> <p>The argument to <code>openForClass:</code> may be any class. It creates a "message set" browser, with two subviews. The top subview (subview A) contains the list of messages in the class' protocol, along with the name of the class in which each message is defined. The bottom subview (subview B) is a code view of the selected message.</p>
<b>how created</b>	Evaluate the code above, then designate the area for the view with red button.
<b>how terminated</b>	Blue button choose <code>close</code> .
<b>blue button activity</b>	Default
<b>red button activity</b>	
subview A	Selects (or deselects) the method to which the cursor points. When one is selected, the bottom subview will display its code.
subview B	Behaves like a standard code editor.
<b>yellow button activity</b>	
subview A	Standard message set menu.
subview B	Standard code editor menu.

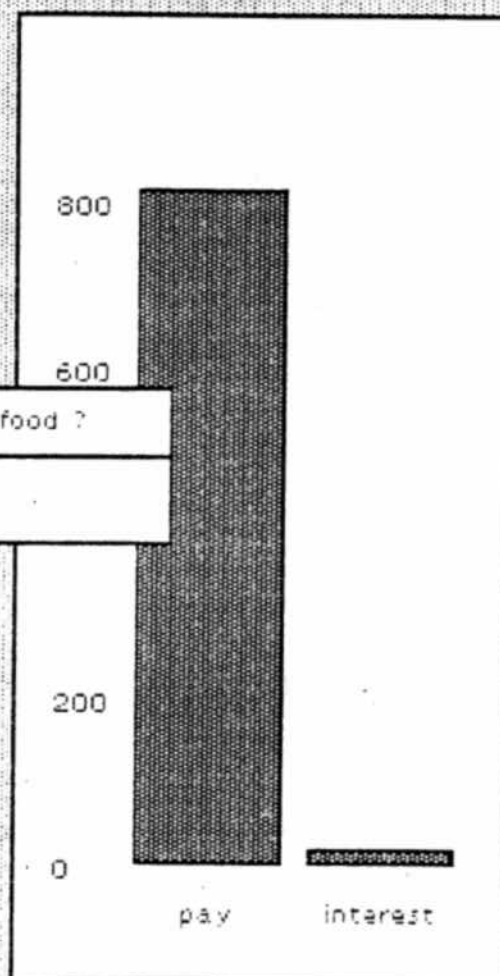
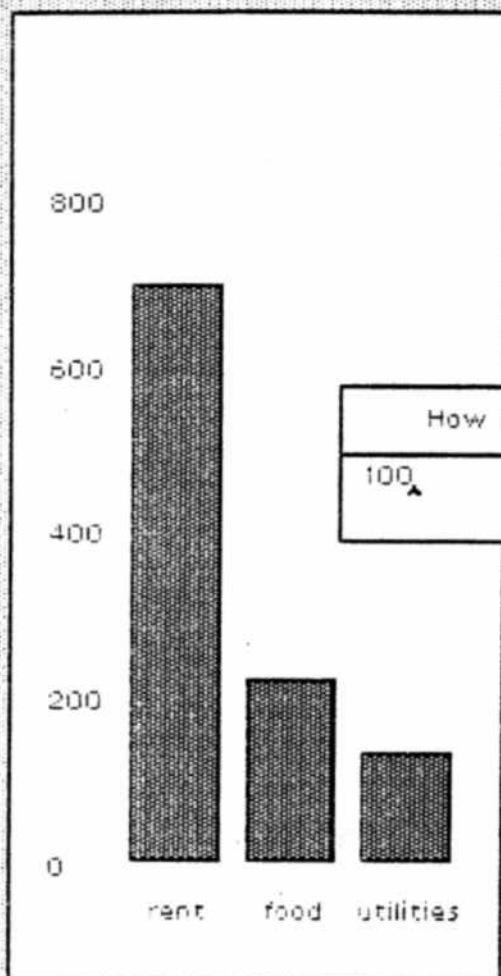




### 3. Project Browser (File 12)

This "goodie" creates a browser to look at all the system projects at the same time. File 12 contains the code for creating a project browser as described in *Smalltalk-80: The Interactive Programming Environment*. It creates a new class `ProjectBrowser` and modifies the class `Project`.

name	Project Browser	
general description	Used to create a browser to view the system projects.	
how accessed	<p>To bring the class definitions into your system, file in File 12 from the tape. The class on this file is categorized under <code>Interface-Projects</code>. To create a view, evaluate the expression</p> <p><code>ProjectBrowser open</code></p> <p>It creates a project browser with two subviews. The top subview (subview A) contains the list of projects in the system. The bottom subview (subview B) is a text view of the selected project.</p>	
how created	Evaluate the code above, then designate the area for the view with red button.	
how terminated	Blue button choose close.	
blue button activity	Default	
red button activity		
subview A	Selects (or deselects) the project to which the cursor points. When one is selected, the bottom subview will display its text.	
subview B	Behaves like a standard paragraph editor.	
yellow button activity		
subview A	enter	Make the selected project be current. (You may want to open another project browser in that project to get back to the original project.)
	add project	Add a new project to the system and to the browser. You will be prompted for a title for the new project that will appear in subview A.
	remove project	Remove the selected project from the system. You may not remove the active project.
subview B	Standard paragraph editor menu.	

**Financial History**

How much for food ?

100

## 4. Financial History (File 13)

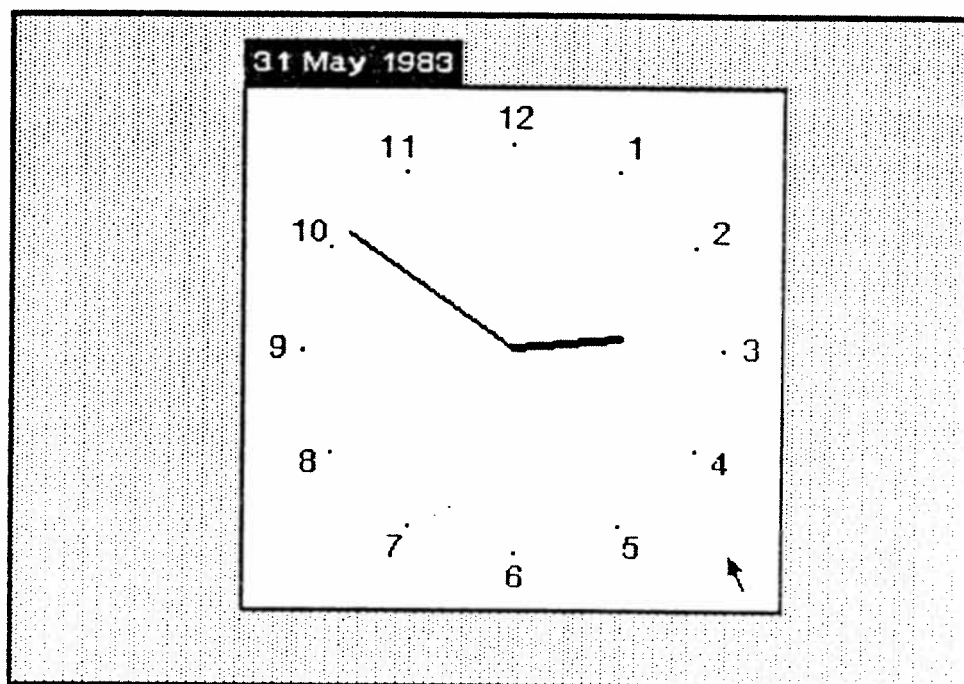
This "goodie" is the financial history example used in *Smalltalk-80: The Interactive Programming Environment*. File 13 includes four class definitions: `FinancialHistory` models activity of a simple budget; `BarChartView` displays views of models (in particular, instances of `FinancialHistory`) as bar charts; `FinancialHistoryView` combines two `BarChartViews` for expenses and incomes; and `FinancialHistoryController` handles user interaction with a `FinancialHistoryView`.

name	Financial History	
general description	Models and allows viewing and editing of a simple budget.	
how accessed	<p>To bring the class definitions into your system, file in File 13 from the tape. The class on this file is categorized under Financial Tools, so you may need to update your browsers by choosing yellow button menu item update. To create a view, evaluate the expression in the method <code>FinancialHistory exampleWorkspace</code></p> <p>Smalltalk at: <code>#HouseholdFinances</code> put: <code>nil</code>.</p> <p>and then evaluate the other expressions in the example method. This creates a view of a financial history that already has some expenses and incomes. The two bar charts in the view show the amounts of expenses and incomes.</p>	
how created	Evaluate the code above, then designate the area for the view with red button.	
how terminated	Blue button choose close.	
blue button activity	Default	
yellow button activity	spend	Add an expense to the financial history. You will be prompted for the name of the expense, then prompted for the amount. If the name is new, a new bar will be added to the chart.
	receive	Add an income to the financial history. You will be prompted for the name of the income, then prompted for the amount. If the name is new, a new bar will be added to the chart.

## 5. Clock (File 14)

This "goodie" is a view of a real-time clock. File 14 includes two class definitions, one which is the "view," `ClockView` which displays the time of day, and the other of which is a "controller," `ClockController` which handles user interaction with a `ClockView`.

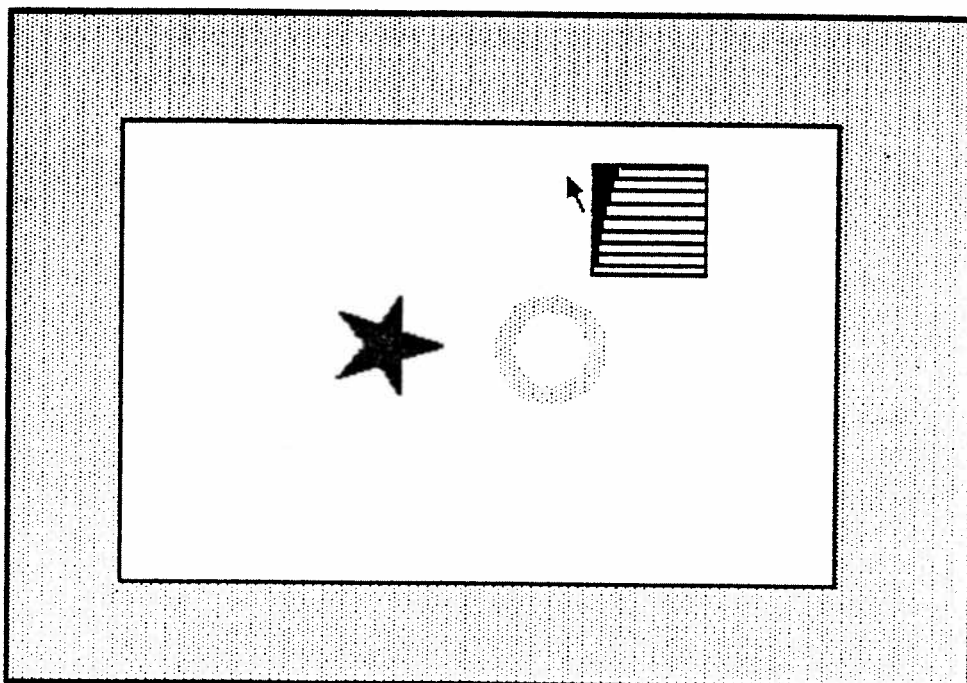
name	<b>Clock</b>
general description	Displays the time of day.
how accessed	<p>To bring the class definitions into your system, file in File 14 from the tape. The class on this file is categorized under Graphics-Clocks, so you may need to update your browsers by choosing yellow button menu item update. To create a view, evaluate the expression</p> <p><code>ClockView open.</code></p> <p>This creates a standard system view whose title is the current date and whose hands are the current time. The clock will refresh itself every minute, so it is best not to put other views on top of it.</p>
how created	Evaluate the code above, then designate the area for the view with red button. Note that the view remains square.
how terminated	Blue button choose close. Note that this is the only way to terminate the refresh of the clock.
blue button activity	Default



## 6. Animation (File 15)

This "goodie" is a set of class definitions that demonstrate animation. File 15 includes seven class definitions that define various animation entities. The example is a simple animation; better animations are welcome.

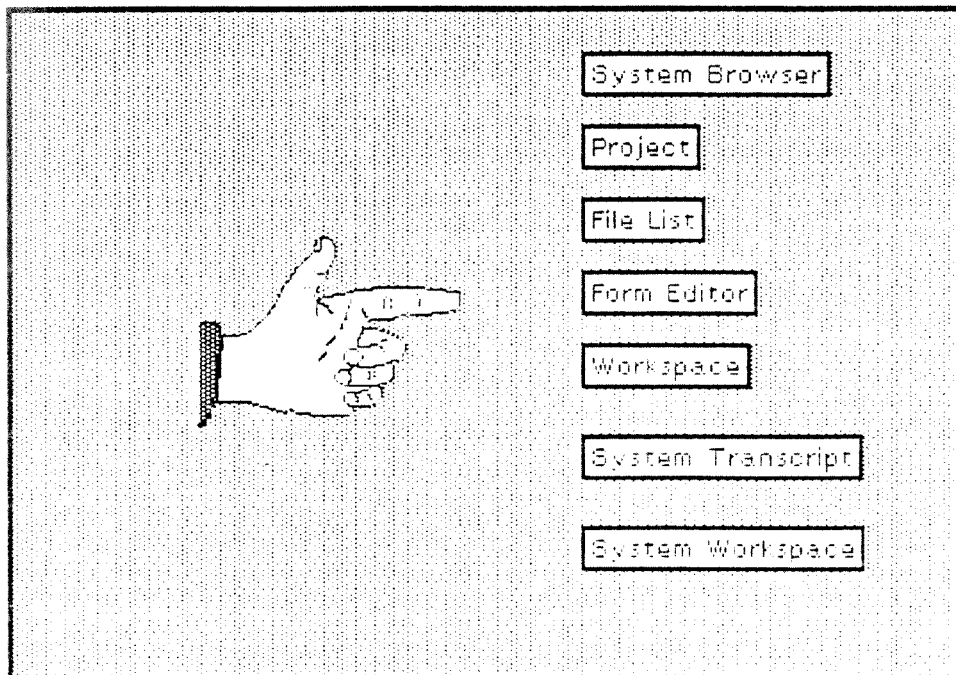
name	Animation
general description	Provides classes that can give animation. The types of animation entities include those that bounce within a window, those made up of a sequence of images (rotating star), those that have transparent as well as opaque areas, and those whose images are created dynamically (e.g., from a rectangle on the screen near the cursor).
how accessed	To bring the class definitions into your system, file in File 15 from the tape. The classes on this file are categorized under Graphics-Animation, so you may need to update your browsers by choosing yellow button menu item update. The last line of the file will start an animation; to restart it, evaluate the expression  WindowNode example
how created	Evaluating the code above creates a box on the display and three animation entities that bounce within that box. One is a "spinning" star, one is a transparent/opaque torus, and the other a box that gets its contents from the area of the display near the cursor. Play with the animation by moving the cursor and seeing how the box changes
how terminated	Press any mouse button while the cursor is in the view.



## 7. Let's Have a Hand for that Demo (File 16)

This "goodie" is a large pointing hand, useful for pointing out items on the screen during demonstrations. File 16 includes the code for doing this.

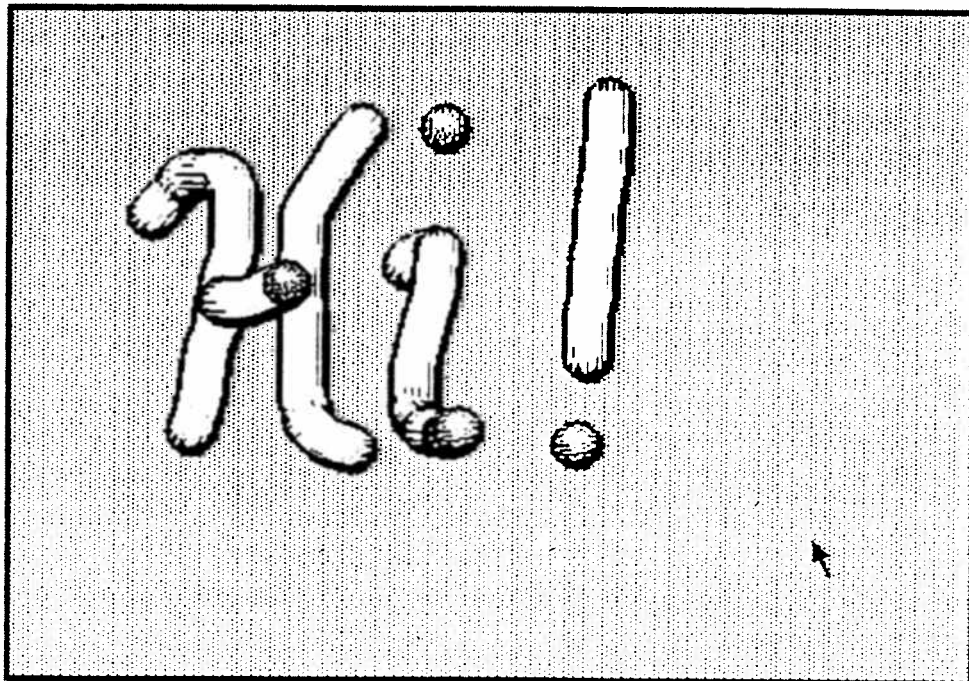
name	Pointing Hand
general description	Creates a large pointing hand that moves with the cursor.
how accessed	To bring the code into your system, file in File 16 from the tape.
how created	Once the code is filed in, any time you are inside a Standard System View and you hold down the left-shift key and then hold down the blue button, the large hand will follow the cursor. You may release the left-shift key any time thereafter and the hand will remain as long as the blue button is down.
how terminated	Release the blue button.



## 8. Toothpaste Graphics (File 17)

This "goodie" is a method added to Form class that draws spheres. The algorithm, by Ward Cunningham of Tektronix, was styled after the work of Ken Knowlton, published in *Computer Graphics*, 15(4)352.

name	Toothpaste Graphics
general description	Draws spheres on the display that resemble squeezed toothpaste.
how accessed	To bring the code into your system, file in File 17 from the tape and evaluate the expression Form toothpaste
how created	Once the code is filed in, and the expression is evaluated, whenever the red button is held down, toothpaste appears at the cursor position.
how terminated	Click the yellow button.



7. 1:

7. 1:

7. 1:



## Part 3: Multiple Inheritance Description

The Smalltalk-80 System includes support for multiple inheritance of classes. We include a paper by Alan Borning and Daniel Ingalls that describes the approach used. Here we will provide a little more background on its use. The status of the implementation provided is similar to a "Goodie" in that it is nowhere actually used in the system itself, and it has not been rigorously tested. Nonetheless, all the necessary support is there, and in many situations the use of multiple superclasses can lead to a cleaner organization.

As an example, consider the Stream hierarchy presently in the system:

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

In this organization, it is necessary for all ReadStream behavior to be copied by the programmer into class ReadWriteStream. Not only is this a burden, but it also presents a pitfall for later modification of the system, for there is no mechanism to maintain the constraint that these two copies of code must remain the same. It would appear more appropriate in this case to have ReadWriteStream be a subclass of *both* ReadStream and WriteStream.

In order to create a class with multiple superclasses, you must type a different expression in the browser in place of the normal class template. To define our new kind of read-write stream, for example, you would type and 'accept'

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

If one were to enter and accept the above definition for NewReadWriteStream, the Transcript would show the following two messages:

```
NewReadWriteStream has conflicting inherited methods
-- consult browser for their names
NewReadWriteStream class has conflicting inherited methods
-- consult browser for their names
```

If you browse to the category 'conflicting inherited methods' in NewReadWriteStream, you will find a bunch of methods with '↑self conflictingInheritanceError' as their code. These methods were automatically generated because the system did not know which of the methods from ReadStream or WriteStream was appropriate to inherit. Typically, you would remove the source of conflict, or replace the generated error code with code which computes an appropriate result.

In this particular case, the conflicts arise for the most part from conflicting styles - many parts of the present system use dummy methods ("subclassResponsibility" and "shouldNotImplement") to document proper protocol. Unfortunately, this style often interferes with simple use of multiple inheritance. As we evolve a clearer style of using multiple inheritance, these issues will be dealt with more effectively.

As an exercise, and as a convenient feature for working with multiple inheritance, you could redefine

Class class>instance creation>template:

so that the class definition template can optionally be in the multiple inheritance form above. The method could, for example, return a template of the above form when the left-shift key is pressed. The decision whether to print the normal or multiple-inheritance pattern is made in

ClassDescription>printing>description,

based on whether a class has multiple superclasses. Therefore, once a class has been defined to have multiple superclasses, its definition will subsequently be printed as above.

You may notice that the multiple inheritance template for classes does not provide for reference to pool dictionaries. The reason is that multiple inheritance provides a simpler mechanism for accomplishing the same effect. Instead of a pool, one simply defines a new class with the desired variables declared as class variables. This class can then be mentioned as a superclass of any class which needs access to the common variables. Thus the generalization of semantics afforded by multiple inheritance eliminates the need for an added mechanism (pool variables) in the non-extended system. An additional benefit to this approach is that the normal class mechanism provides a place (initialization message and comments) to compute initial values for the variables and to document their purpose and use.

It is worth noting that if most behavior is inherited from a single superclasses, that class should be mentioned *first* in the list of superclasses. In this way, fewer methods will need to be copied, and less overhead of consistency maintenance will be incurred.

In the implementation, a distinction is made between the principal (dynamic) superclass and other superclasses. The principal superclass is the first one mentioned in the superclass list of the definition, and it is seen by the virtual machine as a normal Smalltalk-80 superclass. The remaining superclasses are simulated by copying messages and instance variables. As a result, a certain amount of bookkeeping is necessary to maintain the consistency of this simulation when methods are added, removed, and changed.

# Multiple Inheritance in Smalltalk-80

Alan H. Borning  
Computer Science Department, FR-35  
University of Washington  
Seattle, WA 98195

Daniel H. H. Ingalls  
Xerox Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto, CA 94304

## Abstract

Smalltalk classes may be arranged in hierarchies, so that a class can inherit the properties of another class. In the standard Smalltalk language, a class may inherit from only one other class. In this paper we describe an implementation of multiple inheritance in Smalltalk.

## 1. Introduction

Smalltalk is a powerful interactive language based on the idea of objects that communicate by sending and receiving messages [Ingalls 78, LRG 81, Goldberg 82]. Every Smalltalk object is an instance of some class. Classes are organized hierarchically, so that a new class is normally defined as a subclass of an existing class. The subclass inherits the instance storage requirements and message protocol of its superclass. It may add new information of its own, and may override inherited responses to messages.

In standard Smalltalk, a class can be a subclass of only a single superclass. On occasion, this restriction is undesirable and leads to unnatural coding styles. For example, the Smalltalk system includes a class *Transcript* that displays and records notification messages and the like. It is declared to be a subclass of *Window*, but also has the message protocol of a *WriteStream* to which one can append characters. Since it cannot be a subclass of both *Window* and *WriteStream*, the necessary methods for stream behavior must all be duplicated in *Transcript*. Such duplication is unmodular. If some method for streams is added or modified, the class *Transcript* does not automatically feel this change (as it would if it were a subclass of *WriteStream*).

The natural solution is to allow classes to be subclasses of more than one superclass. In this paper we describe an implementation of multiple superclasses, which is now available in the Smalltalk-80 system used within Xerox PARC.

## 2. Semantics of Multiple Superclasses

A class may have any number of superclasses; however, an instance is always an instance of precisely one class.

### 2.1. Message Handling

When an instance receives a message, it first checks the method dictionary of its own class for a method for receiving that message. If none is found, it searches the method dictionaries of its immediate superclasses, then *their* superclasses, and so on. If a single method is found, then it is run. If no method or more than one method is found, an error message is issued. The overriding of inherited methods is still allowed; it is an error only if a class with no method of its own inherits different methods from two or more of its superclasses. Further, it is not an error if the same method is inherited via several paths. (This is a simplified explanation; Section 4 describes our actual implementation.)

### 2.2. Access to Overridden Inherited Methods

In single-superclass Smalltalk, the programmer can access an inherited overridden method using the reserved word *super*. For example, in code defined in a given class *C*, the inherited method for *copy* may be invoked using the expression *super copy*, even if *C* itself has a method for *copy*.

This mechanism may be insufficient in the presence of multiple superclasses -- for example, if *C* inherits two different methods for *copy*, the user needs a way to indicate which is wanted. To allow for this, we extend the syntax of Smalltalk by adding *compound selectors* consisting of a class name, followed by a period, followed by the actual selector, e.g. *Object.copy*. When one of these compound selectors is used in a message, the lookup for the method starts with the class named in the compound selector.

When there is no ambiguity, it is still convenient to be able to say "use the method inherited from my superclass" without naming that superclass. In analogy with the above form of compound selector, this can be accomplished by writing e.g. *self super.copy*.

Finally, there are times when one would like to invoke *all* the inherited methods for a given selector, rather than just one of them; the principal example of this is for the *initialize* method. To accomplish this, the programmer would write *self all.initialize*. It would be straightforward to add other sorts of method combination schemes using this basic mechanism.

### 3. Examples of Using Multiple Inheritance

In this section we present a number of examples that illustrate the usefulness of multiple inheritance.

#### 3.1. Simula-style Linked Lists

Simula, which has a single-superclass inheritance hierarchy, defines a list-processing package that supports doubly-linked lists [Birtwistle 73]. The class *Link* specifies that each of its instances contain a reference to a successor and to a predecessor object. Subclasses of *Link* may then be defined that inherit this ability to be included in linked lists. An analogous class may be easily defined in Smalltalk. (An advantage of implementing linked lists in this way, rather than having a separate link object that simply points to an object X in the list, is that X can know about the list in which it resides.)

However, there is a problem with the class *Link* in both Simula and single-superclass Smalltalk. Given an arbitrary existing class C, unless C already has *Link* in its superclass hierarchy, a programmer cannot use C in defining a new subclass that also has the properties of a *Link*.

Multiple superclasses provide a natural solution. For example, if the programmer wants to make objects that are like windows and can also be included in doubly-linked lists, he or she can simply define a new class *QueueableWindow* that is a subclass of both *Window* and *Link*. The new class will inherit the instance state requirements and message protocol of both *Window* and *Link*, yielding the desired behavior.

#### 3.2. Other Examples

As mentioned in the introduction, another situation in which multiple inheritance is useful is in defining the class *Transcript* as a subclass of *Window* and of *WriteStream*.

To take another example from the standard Smalltalk-80 system, a number of kinds of streams are defined, including *ReadStream*, *WriteStream*, and *ReadWriteStream*. *ReadWriteStream* is rather arbitrarily declared to be a subclass of *WriteStream*, with the extra methods needed for *ReadStream* behavior copied by the programmer. Using our new system, *ReadWriteStream* is naturally defined as a subclass of both *ReadStream* and *WriteStream*.

#### 3.3. Pool Variables

This last example is of a somewhat different nature. In addition to instance variables, the Smalltalk-80 language allows the programmer to define *class variables* that are shared by all instances of a given class and its subclasses. However, on occasion, the programmer wants variables that are to be shared by a number of non-hierarchical classes, but which aren't properly declared to be global variables. A mechanism for handling this exists already: one may declare a dictionary of *pool variables* that may be shared among several classes. (An example of this is the *FilePool* of constants and variables that are shared by all the classes used in file I/O.)

Multiple superclasses provide a more elegant solution. Rather than using pool variables, one can for example define a class *FileObject* that has class variables corresponding to all the variables that used to be in *FilePool*. Each of the file classes can now be made a subclass of *FileObject* as well as of its old superclass, so that it has access to these shared variables. In this way, the pool mechanism becomes unnecessary and could be eliminated from the language.

### 4. Implementation

#### 4.1. Finding the Right Method to Receive a Message

Our implementation of multiple inheritance is a compromise between the extremes of strict runtime method lookup and copying down inherited methods from all superclasses.

In the standard Smalltalk-80 system, methods inherited from superclasses are looked up dynamically. This has the advantage that the system is not cluttered with copied methods, and that there are no copies to update when a method is edited. An alternative would be to copy the inherited methods down into each subclass. This would make finding the methods easier at runtime, at the expense of greater code size and updating complexity.

In our implementation of multiple inheritance, the standard dynamic lookup scheme is used for methods on the chain consisting of the first superclass of each class. If a class C has more than one superclass, at the time C is created it checks each message to which it can respond. If the appropriate method would be found by the dynamic lookup, nothing is done. However, if the appropriate method is in some other superclass, then the code for that method is recompiled in C's method dictionary, so that it will be found at run time.

Finally, if there are conflicting inherited methods for a given selector, an error method is compiled in C for that selector and the user is notified. These error methods are put into a special category, making it easy for the user to browse to them and to resolve the conflicts as necessary.

#### 4.2. Implementation of Compound Selectors

As described in Section 2.2, the programmer can access inherited methods using constructs such as *self Object.copy*, *self super.copy*, and *self all.initialize*. To implement these extensions, we changed the Smalltalk parser to treat compound selectors as single symbols, so that the code that is compiled in C for e.g. *self B.copy* actually sends the selector *B.copy*. The first time this is executed, no method for *B.copy* will be found. When this occurs, the interpreter invokes *Object messageNotUnderstood*. The usual behavior at this point is to bring up an error window. However, we modified *Object messageNotUnderstood* to first check for compound selectors. If one is found, then the system attempts to compile an appropriate method for that compound selector by first verifying that *B* is a superclass of *C*, and then looking for a *copy* method in *B* or its superclasses. If one is found, that method is recompiled in *C* under the selector *B.copy*. The system then resends the message, whereupon it will find the newly inserted method. The next time *B.copy* is sent, this method will be found, making the operation efficient. Selectors such as *super.copy* and *all.initialize* are handled by the same mechanism.

#### 4.3. Instance State

A subclass inherits all the instance field requirements of its superclasses, and can specify additional fields of its own. There is only one copy of fields inherited from a superclass via two inheritance paths. In our current implementation, it is an error if there are different inherited instance fields with the same name. (One of our previous experimental implementations [Borning 80] included a mechanism similar to the compound selector construct that allowed the programmer to disambiguate conflicting field names. We may re-introduce this mechanism if the present restriction proves too burdensome.)

To access or store into instance fields, the bytecodes produced by the Smalltalk compiler include instructions such as "load instance field 1". It is of course essential that code inherited from superclasses use the correct field positions for the subclass. Our scheme takes care of this in the following manner. The instance fields are arranged so that the fields inherited from the superclasses on the dynamic lookup chain have the same positions as they do in the superclasses. (This is the same situation as in single-superclass Smalltalk.) In general, fields inherited from other superclasses won't be in the same positions, but when the code for methods from these other superclasses is recompiled into the new subclass, the field positions are adjusted appropriately. As an optimization, before recompiling a method from a superclass the system checks if the offsets of all the fields it references are the same in the subclass. If this is the case, then the system simply copies a pointer to the original method, rather than recompiling it.

#### 4.4. Dynamic Updating

In the Smalltalk environment, the user can add, delete, and edit methods incrementally, and then immediately make use of the changed code. In our multiple-inheritance implementation, some updating may be necessary when such changes are made. If a method is edited which has been recompiled or copied into some subclasses, then the newly edited method is recompiled or copied into subclasses as necessary. Similarly, if a method is added or deleted, it may affect which inherited method should be used, and may require changes in the copied inherited methods. Again, the system takes care of this updating automatically.

If methods with compound selectors (e.g. *super.printOn:*) have been automatically compiled into some subclasses, then these methods may be invalid as well. Each such method that may no longer be valid is simply deleted; as described above, it will be recompiled automatically the first time a message is sent that invokes it.

#### 4.5. A Note on the Implementation Process

The changes required to add multiple inheritance to Smalltalk-80 are only a few pages of Smalltalk code. For example, changing the Smalltalk syntax to allow compound selectors of the form *Point.copy* or *Point. +* required a change to only one method. Moreover, no changes to the Smalltalk-80 virtual machine were required. There are few other programming environments in which such a fundamental extension could be made so easily.

### 5. Relation to other Work

A number of other systems have used multiple inheritance. Among the systems implemented in Smalltalk, the constraint laboratory ThingLab [Borning 81] and the PIE knowledge representation language [Goldstein and Bobrow 80] both supported multiple inheritance. The authors have also implemented some experimental predecessors of the present system [Borning 80].

Some extensions to Lisp allow the use of similar object-oriented programming techniques. The "flavors system" in MIT Lisp Machine Lisp [Cannon 80] allows an object to be defined using several flavors (analogous to multiple superclasses); this system also contains an extensive repertoire of method combination techniques for combining inherited information. Another object-oriented Lisp extension with multiple inheritance is the LOOPS system [Bobrow and Stefik 82], implemented in Interlisp.

The Traits system [Curry 82], imbedded in the Mesa system, is yet another multiple inheritance implementation. It has received extensive use in the coding of the Xerox Star office information system.

## References

- [Birtwistle 73] Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K.  
*SIMULA Begin*.  
Auerbach Press, 1973.
- [Bobrow and Stefik 82] Bobrow, D.G., and Stefik, M.J.  
LOOPS: An Object Oriented Programming System for Interlisp.  
1982.
- [Borning 80] Borning, A.H.  
Multiple Inheritance in Smalltalk.  
1980.  
Unpublished report, Learning Research Group, Xerox PARC.
- [Borning 81] Borning, A.H.  
The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory.  
*ACM Transactions on Programming Languages and Systems* 3(4):353-387, October, 1981.
- [Cannon 80] Cannon, H.I.  
*Flavors*.  
Technical Report, MIT Artificial Intelligence Lab, 1980.
- [Curry 82] Curry, G., Baer, L., Lipkie, D., and Lee, B.  
Traits: An Approach to Multiple Inheritance Subclassing.  
In *ACM-SIGOA Conference on Office Automation Systems*. ACM, June, 1982.
- [Goldberg 82] Goldberg, A.J., Robson, D., and Ingalls, D.H.H.  
Smalltalk-80: The Language and its Implementation.  
1982.  
Forthcoming book.
- [Goldstein and Bobrow 80] Goldstein, I.P., and Bobrow, D.G.  
Extending Object Oriented Programming in Smalltalk.  
In *Proceedings of the Lisp Conference*.  
Stanford University, 1980.
- [Ingalls 78] Ingalls, D.H.H.  
The Smalltalk-76 Programming System: Design and Implementation.  
In *Proceedings of the Fifth Annual Principles of Programming Languages Symposium*, pages 9-16. ACM, January, 1978.
- [LRG 81] The Xerox Learning Research Group.  
The Smalltalk-80 System.  
*Byte* 6(8):36-48, August, 1981.

## Part 4: Errata in *Smalltalk-80: The Language and its Implementation*

There are some known errors in the Virtual Machine Specification in Part Four of *Smalltalk-80: The Language and its Implementation*. Please let us know if you find any others.

The following pages list the errors in the second printing which has only the line "*Reprinted with corrections, May 1983*" on the Library of Congress page. The first printing does not have this line (but does have the same errors). The third printing, which should have a second "*Reprinted with...*" line, should have these corrected. In addition to the errors noted on the following pages, there is one extra error in the first printing, and three in the first, second, and possibly the third.

1. (pp. 612-616). (First printing only) Two asterisks discriminating required from optional primitives are in the wrong place. The affected primitives should read:

<i>Primitive Index</i>	<i>Class-Selector</i>	<i>Optional/Required</i>
12*	SmallInteger //	Optional
13	SmallInteger quo:	Required

2. (p. 632) **primitiveAtEnd** is specified as testing whether `index >= length`. This test is not necessary.
3. (p. 636) **primitiveAsObject** sends the message **hasObject:**, which is not defined. Its definition is a check to see whether the argument `oop` is a valid object. For example, if your memory system has zero for a reference count of free chunks and free oops, and non-zero for valid objects, then this is just a test for the reference count being zero.
4. (p. 637) **primitiveNewMethod** is defined to initialize the literal frame of the method to 0 it should initialize it to nil. This code could be in **primitiveNewMethod**, or be a change to **allocate:odd:pointer:extra:class:** (p. 685).

## Image Bugs

Peter Lee

Dept. E.E.C.S., U.C. Berkeley

Code for the class method "fromString:" in class String now reads as follows:

```
!String class methodsFor: 'instance creation'!  
  
fromString: aString  
    "Answer a new String that is a copy of the argument, aString."  
  
    / newString/  
    newString ← self new: aString size.  
    aString size do:  
        [:i /newString at: i put: (aString at: i)].  
    ↑ newString
```

it should be:

```
!String class methodsFor: 'instance creation'!  
  
fromString: aString  
    "Answer a new String that is a copy of the argument, aString."  
  
    / newString/  
    newString ← self new: aString size.  
    1 to: aString size do:  
        [:i /newString at: i put: (aString at: i)].  
    ↑ newString
```



There is a small discrepancy between the specification of the object-memory in Chapter 30 and Trace 1 (all memory references) provided with the licenses. Scattered throughout this trace are a few lines that represent memory references that the object memory specified in Chapter 30 would not make. For example, the trace of the first return bytecode looks like

```
↑(method) of false
  11048 pointer: 1 = 293
  38738 pointer: 0 ← 2
  38738 pointer: 1 ← 2
  22 pointer: 2 = 57357 *
  11048 pointer: 3 = 27492
  11048 pointer: 5 = 25286
  11048 pointer: 3 = 27492
  11048 pointer: 1 = 293
  11048 pointer: 2 = 9
  11048 pointer: 10 ← 4
  27492 byte: 145 = 107
```

The memory reference represented by the line with the asterisk (and similar lines are probably found in similar places throughout Trace 1) is not necessary.

The traces are made by a Smalltalk-80 implementation of the virtual machine. The bytecode interpreter part of that implementation is identical to the code in Chapters 27, 28, and 29; however, the object memory part differs from the code shown in Chapter 30. In particular, it determines whether an object contains pointers by loading the instance specification from the object's class. This happens in the recursive freeing routine which is often invoked in a return bytecode. The code in Chapter 30, on the other hand, looks at the "contains pointers" bit in the object table entry for the object, which is more efficient. The spurious lines identify the unnecessary memory access for the class' instance specification.

Thanks to Ben Cutler and Chris Hudak at Yale for pointing out this discrepancy, and Dave Robson for identifying its source.

## Blue Book Bugs

Geoffrey Poole  
International Technical Group, Citibank , Brussels

Page 681 in method forAllOtherObjectsAccessibleFrom:suchThat:do:.

```
size < HugeSize
  ifTrue: [...]
  ifFalse: [self heapChunkOf: current
                    word: size + 1 put: offset].
should be
```

```
size < HugeSize
  ifTrue: [...]
  ifFalse: [self heapChunkOf: current
                    word: size put: offset].
```

otherwise, the word indexed is not the "extra" word but the length word of the next object

page 84 '0' => '350'  
 page 181 remove repeated definition of 'location:'  
 page 555 'unary' => 'binary'  
 page 556 'unary' => 'binary'  
 page 557 'unary' => 'binary'  
 page 558 'unary' => 'binary'  
 page 559 'unary' => 'binary'  
 page 578 'MethodContext' => 'CompiledMethod' (3 times)  
 page 610 add paragraph about circular structures  
 page 614 'value:' => 'value'  
 page 617 tests for negative and integerValue replaced by extractBits  
 \* includes paste-up  
 page 623 16-bit field value change  
 \* includes paste-up  
 page 630 '2' => '3'  
 page 650 '128' <=> '130'  
 page 657 '6' => '5'  
 page 669 initialize 'objectPointer' and 'LastFreeChunkList' => 'BigSize' (2 times)  
 \* includes paste-up  
 page 686 rearrange conditionals  
 \* includes paste-up  
 page 687 'load' => 'fetch'  
 page 688 initialize 'classPointer'  
 \* includes paste-up  
 page 709 add '586'  
 page 711 '562' => '622'

the search for the response to `initialBalance:` begins in `FinancialHistory` class, i.e., in the class methods for `FinancialHistory`. A method for that selector is found there. The method consists of two messages:

1. Send super the message `new`. 350
2. Send the result of 1 the message `setInitialBalance:` ①

The search for `new` begins in the superclass of `FinancialHistory` class, that is, in `Object` class. A method is not found there, so the search continues up the superclass chain to `Class`. The message selector `new` is found in `Class`, and a primitive method is executed. The result is an uninitialized instance of `FinancialHistory`. This instance is then sent the message `setInitialBalance:`. The search for the response begins in the class of the instance, i.e., in `FinancialHistory` (in the instance methods). A method is found there which assigns a value to each instance variable.

The evaluation of

`FinancialHistory new`

is carried out in a similar way. The response to `new` is found in `FinancialHistory` class (i.e., in the class methods of `FinancialHistory`). The remaining actions are the same as for `initialBalance:` with the exception of the value of the argument to `setInitialBalance:`. The instance creation methods must use `super new` in order to avoid invoking the same method recursively.

## Initialization of Class Variables

The main use of messages to classes other than creation of instances is the initialization of class variables. The implementation description's variable declaration gives the names of the class variables only, not their values. When a class is created, the named class variables are created, but they all have a value of `nil`. The metaclass typically defines a method that initializes the class variables. By convention, the class-variable initialization method is usually associated with the unary message `initialize`, categorized as class initialization.

Class variables are accessible to both the class and its metaclass. The assignment of values to class variables can be done in the class methods, rather than indirectly via a private message in the instance methods (as was necessary for instance variables).

The example `DeductibleHistory`, this time with a class variable that needs to be initialized, is shown next. `DeductibleHistory` is a subclass of `FinancialHistory`. It declares one class variable, `MinimumDeductions`.

## The Drunken Cockroach Problem

We can use some of the collection classes to solve a well-known programming problem. The problem is to measure how long it takes a drunken cockroach to touch each tile of a floor of square tiles which is  $N$  tiles wide and  $M$  tiles long. To slightly idealize the problem: in a given "step" the cockroach is equally likely to try to move to any of nine tiles, namely the tile the roach is on before the step and the tiles immediately surrounding it. The cockroach's success in moving to some of these tiles will be limited, of course, if the cockroach is next to a wall of the room. The problem is restated as counting the number of steps it takes the cockroach to land on *all* of the tiles at least once.

One straightforward algorithm to use to solve this problem starts with an empty Set and a counter set to 0. After each step, we add to the Set the tile that the cockroach lands on and increment a counter of the number of steps. Since no duplication is allowed in a Set, we would be done when the number of elements in the Set reaches  $N \cdot M$ . The solution would be the value of the counter.

While this solves the simplest version of the problem, we might also like to know some additional information, such as how many times each tile was visited. To record this information, we can use an instance of class Bag. The size of the Bag is the total number of steps the cockroach took; the size of the Bag when it is converted to a Set is the total number of distinct tiles touched by the cockroach. When this number reaches  $N \cdot M$ , the problem is solved. The number of occurrences of each tile in the Bag is the same as the number of times the roach visited each tile.

Each tile on the floor can be labeled with respect to its row and its column. The objects representing tiles in the solution we offer are instances of class Tile. A commented implementation of class Tile follows.

open  
space

```

class name      Tile
superclass      Object
instance variable names  location
                                floorArea
instance methods
accessing

location
    "Answer the location of the receiver on the floor."
    flocation
location: aPoint
    "Answer the location of the receiver on the floor."
    flocation
location: aPoint
    "Set the receiver's location on the floor to be the argument, aPoint."
    location = aPoint
  
```

interpreter's state may have to be changed in order to execute a different CompiledMethod in response to this new message. The interpreter's old state must be remembered because the bytecodes after the send must be executed after the value of the message is returned.

The interpreter saves its state in objects called *contexts*. There will be many contexts in the system at any one time. The context that represents the current state of the interpreter is called the *active context*. When a send bytecode in the active context's CompiledMethod requires a new CompiledMethod to be executed, the active context becomes *suspended* and a new context is created and made active. The suspended context retains the state associated with the original CompiledMethod until that context becomes active again. A context must remember the context that it suspended so that the suspended context can be resumed when a result is returned. The suspended context is called the new context's *sender*.

The form used to show the interpreter's state in the last section will be used to show contexts as well. The active context will be indicated by the word Active in its top delimiter. Suspended contexts will say Suspended. For example, consider a context representing the execution of the CompiledMethod for Rectangle rightCenter with a receiver of 100@100 corner: 200@200. The source method for Rectangle rightCenter is

#### rightCenter

↑ self right @ self center y

The interpreter's state following execution of the first bytecode is shown below. The sender is some other context in the system.

Active	
Method for Rectangle rightCenter	
112	push the receiver (self) onto the stack
♦ 208	send a unary message with the selector in the first literal (right)
112	push the receiver (self) onto the stack
209	send the unary message with the selector in the second literal (center)
207	send the unary message with the selector y
187	send the <del>unary</del> message with the selector @
124	return the object on top of the stack as the value of the message (rightCenter)
literal frame	
· #right	
· #center	
Receiver	100@100 corner: 200@200
Arguments	
Temporary Variables	
Stack	100@100 corner: 200@200
Sender	↓

binary  
(same font)

After the next bytecode is executed, that context will be suspended. The object pushed by the first bytecode has been removed to be used as the receiver of a new context, which becomes active. The new active context is shown above the suspended context.

---

Active

---

## Method for Rectangle right

- ♦ 1 push the value of the receiver's second instance variable (corner) onto the stack
- 206 send a unary message with the selector x
- 124 return the object on top of the stack as the value of the message (right)

Receiver 100@100 corner: 200@200

Arguments

Temporary Variables

Stack

Sender ↓

---

Suspended

---

## Method for Rectangle rightCenter

- 112 push the receiver (self) onto the stack
- 208 send a unary message with the selector in the first literal (right)
- ♦ 112 push the receiver (self) onto the stack
- 209 send the unary message with the selector in the second literal (center)
- 207 send the unary message with the selector y
- 187 send the ~~unary~~ message with the selector @
- 124 return the object on top of the stack as the value of the message (rightCenter)

literal frame

#right

#center

Receiver 100@100 corner: 200@200

Arguments

Temporary Variables

Stack

Sender ↓

binary



The next cycle of the interpreter advances the new context instead of the previous one.

---

### Active

---

#### Method for Rectangle right

- 1 push the value of the receiver's second instance variable (corner) onto the stack
- ♦ 206 send a unary message with the selector x
- 124 return the object on top of the stack as the value of the message (right)

Receiver 100@100 corner: 200@200

Arguments

Temporary Variables

Stack 200@200

Sender ↓

---



---

### Suspended

---

#### Method for Rectangle rightCenter

- 112 push the receiver (self) onto the stack
- 208 send a unary message with the selector in the first literal (right)
- ♦ 112 push the receiver (self) onto the stack
- 209 send the unary message with the selector in the second literal (center)
- 207 send the unary message with the selector y
- 187 send the unary message with the selector @
- 124 return the object on top of the stack as the value of the message (rightCenter)

literal frame

#right  
#center

Receiver 100@100 corner: 200@200

Arguments

Temporary Variables

Stack

Sender ↓

---

binary

In the next cycle, another message is sent, perhaps creating another context. Instead of following the response of this new message (x), we

will skip to the point that this context returns a value (to right). When the result of x has been returned, the new context looks like this:

Active	
<b>Method for Rectangle right</b>	
1	push the value of the receiver's second instance variable (corner) onto stack
206	send a unary message with the selector x
♦ 124	return the object on top of the stack as the value of the message (right)
Receiver	100@100 corner: 200@200
Arguments	
Temporary Variables	
Stack	200
Sender	↓

Suspended	
<b>Method for Rectangle rightCenter</b>	
112	push the receiver (self) onto the stack
208	send a unary message with the selector in the first literal (right)
♦ 112	push the receiver (self) onto the stack
209	send the unary message with the selector in the second literal (center)
207	send the unary message with the selector y
187	send the <del>unary</del> message with the selector @
124	return the object on top of the stack as the value of the message (rightCenter)
literal frame	
#right	
#center	
Receiver	100@100 corner: 200@200
Arguments	
Temporary Variables	
Stack	
Sender	↓

binary

The next bytecode returns the value on the top of the active context's stack (200) as the value of the message that created the context (right). The active context's sender becomes the active context again and the returned value is pushed on its stack.



---

**Active**

---

**Method for Rectangle rightCenter**

112 push the receiver (self) onto the stack  
 208 send a unary message with the selector in the first literal (right)  
 112 push the receiver (self) onto the stack  
 209 send the unary message with the selector in the second literal (center)  
 207 send the unary message with the selector y  
 187 send the ~~unary~~ message with the selector @  
 124 return the object on top of the stack as the value of the message (rightCenter)

binary

**literal frame**

#right  
 #center

**Receiver** 100@100 corner: 200@200

**Arguments****Temporary Variables**

**Stack** 200

**Sender** ↓

---

**Contexts**

The contexts illustrated in the last section are represented in the system by instances of MethodContext. A MethodContext represents the execution of a CompiledMethod in response to a message. There is another type of context in the system, which is represented by instances of BlockContext. A BlockContext represents a block in a source method that is not part of an optimized control structure. The compilation of the optimized control structures was described in the earlier section on jump bytecodes. The bytecodes compiled from a nonoptimized control structure are illustrated by the following hypothetical method in Collection. This method returns a collection of the classes of the receiver's elements.

**classes**

self collect: [:element | element class]

**Collection classes** requires 1 temporary variable

112 push the receiver (self) onto the stack  
 137 push the active context (thisContext) onto the stack  
 118 push the SmallInteger 1 onto the stack  
 200 send a single argument message with the selector blockCopy:  
 164,4 jump around the next 4 bytes  
 104 pop the top object off of the stack and store in the first temporary frame location (element)



depth and the number of temporary variables needed is greater than twelve. The smaller MethodContexts have room for 12 and the larger have room for 32.

**largeContextFlagOf: methodPointer**

1self extractBits: 8 to: 8  
of: (self headerOf: methodPointer)

CompiledMethod

note: the font  
should remain the  
same - that is,  
program font

The literal count indicates the size of the MethodContext's literal frame. This, in turn, indicates where the MethodContext's bytecodes start.

**literalCountOf: methodPointer**

1self literalCountOfHeader: (self headerOf: methodPointer)

**literalCountOfHeader: headerPointer**

1self extractBits: 9 to: 14  
of: headerPointer

The object pointer count indicates the total number of object pointers in a MethodContext, including the header and literal frame.

**objectPointerCountOf: methodPointer**

1(self literalCountOf: methodPointer) + LiteralStart

The following routine returns the byte index of the first bytecode of a CompiledMethod.

**initialInstructionPointerOfMethod: methodPointer**

1((self literalCountOf: methodPointer) + LiteralStart) \* 2 + 1

The flag value is used to encode the number of arguments a CompiledMethod takes and whether or not it has an associated primitive routine.

**flagValueOf: methodPointer**

1self extractBits: 0 to: 2  
of: (self headerOf: methodPointer)

The eight possible flag values have the following meanings:

<i>flag value</i>	<i>meaning</i>
0-4	no primitive and 0 to 4 arguments
5	primitive return of self (0 arguments)
6	primitive return of an instance variable (0 arguments)

```

memory increaseReferencesTo: resultPointer.
self returnToActiveContext: contextPointer.
self push: resultPointer.
memory decreaseReferencesTo: resultPointer

```

This routine prevents the deallocation of the result being returned by raising the reference count until it is pushed on the new stack. It could also have pushed the result before switching active contexts. The `returnToActiveContext:` routine is basically the same as the `newActiveContext:` routine except that instead of restoring any cached fields of the context being returned from, it stores nil into the sender and instruction pointer fields.

#### **returnToActiveContext: aContext**

```

memory increaseReferencesTo: aContext.
self nilContextFields.
memory decreaseReferencesTo: activeContext.
activeContext ← aContext.
self fetchContextRegisters

```

#### **nilContextFields**

```

memory storePointer: SenderIndex
  ofObject: activeContext
  withValue: NilPointer.
memory storePointer: InstructionPointerIndex
  ofObject: activeContext
  withValue: NilPointer

```

This paragraph  
needs to be  
added. Words  
in program font  
have been underlined.

Due to the nature of BlockContexts, this implementation of the return bytecodes will create circular structures. Implementations of the object memory that rely exclusively on reference counting to reclaim unused storage will not properly deallocate the objects that make up these circular structures. To avoid this problem, the following additional mechanism should be included. If the active context is a BlockContext and the context being returned to (aContext) is on the sender chain of the active context, the sender pointers of the intervening contexts on the sender chain should be set to nil.

59	
60	LargeNegativeInteger digitAt: LargePositiveInteger digitAt: Object at: Object basicAt:
61	LargeNegativeInteger digitAt:put: LargePositiveInteger digitAt:put: Object basicAt:put: Object at:put:
62	ArrayedCollection size LargeNegativeInteger digitLength LargePositiveInteger digitLength Object basicSize Object size String size
63	String at: String basicAt:
64	String basicAt:put: String at:put:
65*	ReadStream next ReadWriteStream next
66*	WriteStream nextPut:
67*	PositionableStream atEnd
68	CompiledMethod objectAt:
69	CompiledMethod objectAt:put:
70	Behavior basicNew Behavior new Interval class new
71	Behavior new: Behavior basicNew:
72	Object become:
73	Object instVarAt:
74	Object instVarAt:put:
75	Object asOop Object hash Symbol hash
76	SmallInteger asObject SmallInteger asObjectNoFail
77	Behavior someInstance
78	Object nextInstance
79	CompiledMethod class newMethod:header:
80*	ContextPart blockCopy:
81	BlockContext value:value:value: BlockContext valueX (remove the colon) BlockContext value: BlockContext value:value:
82	BlockContext valueWithArguments:
83*	Object perform:with:with:with:



Many of the primitives manipulate integer quantities, so the interpreter includes several routines that perform common functions. The `popInteger` routine is used when a primitive expects a `SmallInteger` on the top of the stack. If it is a `SmallInteger`, its value is returned; if not, a primitive failure is signaled.

**popInteger**

```
| integerPointer |
integerPointer ← self popStack.
self success: (memory isIntegerObject: integerPointer).
self success
    ifTrue: [!memory integerValueOf: integerPointer]
```

*open*

Recall that the `fetchInteger:ofObject:` routine signaled a primitive failure if the indicated field did not contain a `SmallInteger`. The `pushInteger:` routine converts a value to a `SmallInteger` and pushes it on the stack.

**pushInteger: integerValue**

```
self push: (memory integerObjectOf: integerValue)
```

*open*

Since the largest indexable collections may have 65534 indexable elements, and `SmallIntegers` can only represent values up to 16383, primitive routines that deal with indices or sizes must be able to manipulate `LargePositiveIntegers`. The following two routines convert back and forth between 16-bit unsigned values and object pointers to `SmallIntegers` or `LargePositiveIntegers`.

**positive 16BitIntegerFor: integerValue**

```
| newLargeInteger |
newLargeInteger ← memory integerObjectOf: integerValue.
ifTrue: [!memory primitiveFail].
(memory isIntegerValue: integerValue).
ifTrue: [!memory integerObjectOf: integerValue].
newLargeInteger ← memory instantiateClass:
```

ClassLargePositiveIntegerPointer  
withBytes: 2.

```
memory storeByte: 0
    ofObject: newLargeInteger
    withValue: (self lowByteOf: integerValue).
memory storeByte: 1
    ofObject: newLargeInteger
    withValue: (self highByteOf: integerValue).
```

```
!newLargeInteger
```

**positive 16BitValueOf: integerPointer**

```
| value |
(memory isIntegerObject: integerPointer)
```

*(open)*

*(self extractBits: 0 to: 1 of: integerValue) = 0*

*replace this line with this*

*remove these two lines*

*note: the font should remain the same (program font) The new line should have the same horizontal position as the line it replaces. The vertical space created by deleting two lines can be taken up at any of the points indicated*

*See attached paste-up*

**subscript: array with: index**

```

| class value |
class — memory fetchClassOf: array.
(self isWords: class)
  ifTrue: [(self isPointers: class)
    ifTrue: [!memory fetchPointer: index — 1
      ofObject: array]
    ifFalse: [value — memory fetchWord: index — 1
      ofObject: array.
      !self positive16BitIntegerFor: value]]
  ifFalse: [value — memory fetchByte: index — 1
    ofObject: array.
    !memory integerObjectOf: value]

```

**subscript: array with: index storing: value**

```

| class v |
class — memory fetchClassOf: array.
(self isWords: class)
  ifTrue: [(self isPointers: class)
    ifTrue: [!memory storePointer: index — 1
      ofObject: array
      withValue: value]
    ifFalse: [v — self positive16BitValueOf: value.
      self success ifTrue:
        [!memory
          storeWord: index — 1
          ofObject: array
          withValue: v]]]
  ifFalse: [self success: (memory isIntegerObject: value).
    self success ifTrue:
      [!memory storeByte: index — 1
        ofObject: array
        withValue: (self lowByteOf:
          (memory integerValueOf:
            value))]

```

The primitiveAt and primitiveAtPut routines simply fetch or store one of the indexable fields of the receiver. They fail if the index is not a SmallInteger or if it is out of bounds.

**primitiveAt**

```

| index array arrayClass result |
index — self positive16BitValueOf: self popStack.
array — self popStack.
arrayClass — memory fetchClassOf: array.
self checkIndexableBoundsOf: index
  in: array.

```

Note: I cut and pasted a new version of this page. The only further changes are closing the space indicated and distributing it to any of the three locations indicated by "open".

open

close up

open

actually stores 8-bit numbers in byte-indexable fields, but it communicates through the `at:` and `at:put:` messages with instances of `Character`. A `Character` has a single instance variable that holds a `SmallInteger`. The value of the `SmallInteger` returned from the `at:` message is the byte stored in the indicated field of the `String`. The `primitiveStringAt` routine always returns the same instance of `Character` for any particular value. It gets the `Characters` from an `Array` in the object memory that has a guaranteed object pointer called `characterTablePointer`.

#### **primitiveStringAt**

```
| index array ascii character |
index ← self positive16BitValueOf: self popStack.
array ← self popStack.
self checkIndexableBoundsOf: index
    in: array.
self success
    ifTrue: [ascii ← memory integerValueOf: (self subscript: array
        with: index).
        character ← memory fetchPointer: ascii
            ofObject: CharacterTablePointer].
self success
    ifTrue: [self push: character]
    ifFalse: [self unPop: 2]
```

#### **initializeCharacterIndex**

```
CharacterValueIndex ← 0
```

The `primitiveStringAtPut` routine stores the value of a `Character` in one of the receiver's indexable bytes. It fails if the second argument of the `at:put:` message is not a `Character`.

#### **primitiveStringAtPut**

```
| index array ascii character |
character ← self popStack.
index ← self positive16BitValueOf: self popStack.
array ← self popStack.
self checkIndexableBoundsOf: index
    in: array.
self success: (memory fetchClassOf: character) = ClassCharacterPointer.
self success
    ifTrue: [ascii ← memory fetchPointer: CharacterValueIndex
        ofObject: character.
        self subscript: array
            with: index
            storing: ascii].
self success
    ifTrue: [self push: character]
    ifFalse: [self unPop: 2]
```



own, but that change the value of the other keys. The keys on a decoded keyboard only indicate their down transition, not their up transition. On an undecoded keyboard, the standard keys produce parameters that are the ASCII code of the character on the keytop *without* shift or control information (i.e., the key with "A" on it produces the ASCII for "a" and the key with "2" and "@" on it produces the ASCII for "2"). The other standard keys produce the following parameters.

<i>key</i>	<i>parameter</i>
backspace	8
tab	9
line feed	10
return	13
escape	27
space	32
delete	127

For an undecoded keyboard, the meta keys have the following parameters.

<i>key</i>	<i>parameter</i>
left shift	136
right shift	137
control	138
alpha-lock	139

For a decoded keyboard, the full shifted and "controlled" ASCII should be used as a parameter and successive type 3 and 4 words should be produced for each keystroke.

The remaining bi-state devices have the following parameters.

<i>key</i>	<i>parameter</i>
left or top "pointing device" button	128 130
center "pointing device" button	129
right or bottom "pointing device" button	130 128
keyset paddles right to left	131 through 135

segment: s word: w bits: firstBitIndex to: lastBitIndex  
Return bits firstBitIndex to lastBitIndex of word w of segment s.

segment: s word: w bits: firstBitIndex to: lastBitIndex put: value  
Store value into bits firstBitIndex to lastBitIndex of word w of segment s; return value.

When it is necessary to distinguish the two bytes of a word, the left (more significant) byte will be referred to with the index 0 and the right (less significant) byte with the index 1. The most significant bit in a word will be referred to with the index 0 and the least significant with the index 15. Note that self is an instance of class RealObjectMemory in all routines of this chapter.

The most important thing about any implementation of the object memory is that it conform to the functional specification of the object memory interface given in Chapter 27. This chapter describes a range of possible implementations of that interface. In particular, simple versions of some routines are presented early in the chapter and refined versions are presented later as the need for those refinements becomes clear. These preliminary versions will be flagged by including the comment, "\*\*\*Preliminary Version\*\*\*", on the first line of the routine.

## Heap Storage

In a real-memory implementation of Smalltalk, all objects are stored in an area called the *heap*. A new object is created by obtaining space to store its fields in a contiguous series of words in the heap. An object is destroyed by releasing the heap space it occupied. The format of an allocated object in the heap is shown in Figure 30.1. The actual data of the object are preceded by a two-word *header*. The size field of the header indicates the number of words of heap that the object occupies, including the header. It is an unsigned 16-bit number, and can range from 2 up to 65,535.

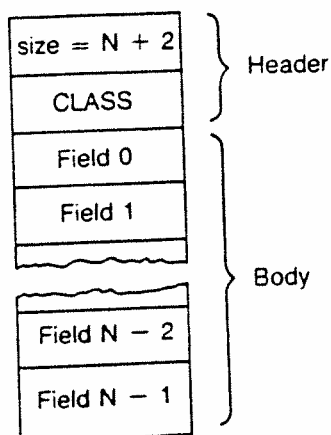


Figure 30.1

**attemptToAllocateChunk: size**

```

| objectPointer |
objectPointer ← self attemptToAllocateChunkInCurrentSegment: size.
objectPointer isNil ifFalse: [objectPointer].
1 to: HeapSegmentCount do:
    [ |
      currentSegment ← currentSegment + 1.
      currentSegment > LastHeapSegment
        ifTrue: [currentSegment ← FirstHeapSegment].
      self compactCurrentSegment.
      objectPointer
        ← self attemptToAllocateChunkInCurrentSegment: size.
      objectPointer isNil ifFalse: [objectPointer]].
    ]

```

close  
up

The attemptToAllocateChunkInCurrentSegment: routine searches the current heap segment's free-chunk lists for the first chunk that is the right size or that can be subdivided to yield a chunk of the right size. Because most objects are smaller than BigSize and most allocation requests can be satisfied by recycling deallocated objects of the desired size, most allocations execute only the first four lines of the routine.

close  
up

**attemptToAllocateChunkInCurrentSegment: size**

```

| objectPointer predecessor next availableSize excessSize newPointer |
size < BigSize
  ifTrue: [objectPointer ← self removeFromFreeChunkList: size].
  objectPointer notNil
    ifTrue: [objectPointer]. "small chunk of exact size handy so use it"
    predecessor ← NonPointer.
    "remember predecessor of chunk under consideration"
    objectPointer ← self headOfFreeChunkList: lastFreeChunkList BigSize
      inSegment: currentSegment.
    "the search loop stops when the end of the linked list is encountered"
    [objectPointer = NonPointer] whileFalse:
      [availableSize ← self sizeBitsOf: objectPointer.
       availableSize = size
        ifTrue: "exact fit — remove from free chunk list and return"
          [next ← self classBitsOf: objectPointer.
           "the link to the next chunk"
           predecessor = NonPointer
            ifTrue: "it was the head of the list; make the next item the head"
              [self headOfFreeChunkList: lastFreeChunkList BigSize
                inSegment: currentSegment put: next]
            ifFalse: "it was between two chunks; link them together"
              [self classBitsOf: predecessor
               put: next].
            objectPointer].

```

objectPointer ← nil.

see attached  
paste up

**CompiledMethods**

A CompiledMethod is an anomaly for the memory manager because its data are a mixture of 16-bit pointers and 8-bit unsigned integers. The only change needed to support CompiledMethods is to add to lastPointerOf: a computation similar to that in the bytecode interpreter's routine bytecodeIndexOf:. MethodClass is the object table index of CompiledMethod.

**lastPointerOf: objectPointer**

```

| methodHeader |
(self pointerBitOf: objectPointer) = 0
  ifTrue:
    [(self classBitsOf: objectPointer) = MethodClass
      ifTrue: [methodHeader ← self heapChunkOf: objectPointer
        word: HeaderSize.
          tHeaderSize + 1 + ((methodHeader bitAnd: 126)
            bitShift: -1)]
      ifFalse: [tHeaderSize]]
    ifFalse:
      [tself sizeBitsOf: objectPointer]

```

This is how  
the page should  
look.

**Interface to the  
Bytecode  
Interpreter**

The final step in the implementation of the object memory is to provide the interface routines required by the interpreter. Note that fetchClassOf: objectPointer returns IntegerClass (the object table index of SmallInteger) if its argument is an immediate integer.

**object pointer access****fetchPointer: fieldIndex ofObject: objectPointer**

```
tself heapChunkOf: objectPointer word: HeaderSize + fieldIndex
```

**storePointer: fieldIndex****ofObject: objectPointer****withValue: valuePointer**

```

| chunkIndex |
chunkIndex ← HeaderSize + fieldIndex.
self countUp: valuePointer.
self countDown: (self heapChunkOf: objectPointer word: chunkIndex).
tself heapChunkOf: objectPointer word: chunkIndex put: valuePointer

```

**word access****fetchWord: wordIndex ofObject: objectPointer**

```
tself heapChunkOf: objectPointer word: HeaderSize + wordIndex
```

**storeWord: wordIndex****ofObject: objectPointer****withValue: valueWord**

```

tself heapChunkOf: objectPointer word: HeaderSize + wordIndex
  put: valueWord

```

byte access

**fetchByte: byteIndex ofObject: objectPointer**

!self heapChunkOf: objectPointer byte: (HeaderSize\*2 + byteIndex)

**storeByte: byteIndex**

**ofObject: objectPointer**

**withValue: valueByte**

!self heapChunkOf: objectPointer

byte: (HeaderSize\*2 + byteIndex)

put: valueByte

reference counting

**increaseReferencesTo: objectPointer**

self countUp: objectPointer

**decreaseReferencesTo: objectPointer**

self countDown: objectPointer

class pointer access

**fetchClassOf: objectPointer**

(self isIntegerObject: objectPointer)

ifTrue: [!IntegerClass]

ifFalse: [!self classBitsOf: objectPointer]

length access

**fetchWordLengthOf: objectPointer**

!(self sizeBitsOf: objectPointer)-HeaderSize

**fetchByteLengthOf: objectPointer**

!(self ~~len~~ WordLengthOf: objectPointer)\*2 - (self oddBitOf: objectPointer)

*fetch*

object creation

**instantiateClass: classPointer withPointers: length**

| size extra |

size ← HeaderSize + length.

extra ← size < HugeSize ifTrue: [0] ifFalse: [1].

!self allocate: size odd: 0 pointer: 1 extra: extra class: classPointer

**instantiateClass: classPointer withWords: length**

| size |

size ← HeaderSize + length.

!self allocate: size odd: 0 pointer: 0 extra: 0 class: classPointer

**instantiateClass: classPointer withBytes: length**

| size |

size ← HeaderSize + ((length + 1)/2).

!self allocate: size odd: length\2 pointer: 0 extra: 0 class: classPointer



instance enumeration

**initialInstanceOf: classPointer**

0 to: ObjectTableSize-2 by: 2 do:

[:pointer |

(self freeBitOf: pointer)=0

ifTrue: [(self fetchClassOf: pointer)=classPointer  
ifTrue: [1pointer]]].

tNilPointer

**instanceAfter: objectPointer**

| classPointer |

ObjectPointer to: ObjectTableSize-2 by: 2 do:

[:pointer |

(self freeBitOf: pointer)=0

ifTrue: [(self fetchClassOf: pointer)=classPointer  
ifTrue: [1pointer]]].

tNilPointer

pointer swapping

**swapPointersOf: firstPointer and: secondPointer**

| firstSegment firstLocation firstOdd |

firstSegment ← self segmentBitsOf: firstPointer.

firstLocation ← self locationBitsOf: firstPointer.

firstPointer ← self pointerBitOf: firstPointer.

firstOdd ← self oddBitOf: firstPointer.

self segmentBitsOf: firstPointer put: (self segmentBitsOf: secondPointer).

self locationBitsOf: firstPointer put: (self locationBitsOf: secondPointer).

self pointerBitOf: firstPointer put: (self pointerBitOf: secondPointer).

self oddBitOf: firstPointer put: (self oddBitOf: secondPointer).

self segmentBitsOf: secondPointer put: firstSegment.

self locationBitsOf: secondPointer put: firstLocation.

self pointerBitOf: secondPointer put: firstPointer.

self oddBitOf: secondPointer put: firstOdd

integer access

**integerValueOf: objectPointer**

tobjectPointer/2

**integerObjectOf: value**

t(value bitShift: 1) + 1

**isIntegerObject: objectPointer**

t(objectPointer bitAnd: 1) = 1

**isIntegerValue: valueWord**

tvalueWord &lt;= -16384 and: [valueWord &gt; 16834]

(note: the "c" at the first of the new line should align vertically with the "o" at the first of the line below) See paste-up attached.

classPointer ← self  
fetchClassOf: objectPointer.

close  
up  
to  
m  
space

- used on, 618-619, 627-632, 635, 639-641, 653, 688
- atchContextRegisters, 584
  - defined on, 583
  - used on, 585, 610
- atchInteger:ofObject:
  - defined on, 574
  - used on, 582-583, 608, 619, 631-632, 643, 646
- atchPointer:ofObject:
  - defined on, 571, 686
- atchWord:ofObject:
  - defined on, 571, 686
- atchWordLengthOf:
  - defined on, 572, 687
  - used on, 627, 638-639, 641, 645
- idIndexOf:
  - defined on, 579
  - used on, 620
- idNewMethodInClass:
  - defined on, 605
  - used on, 605
- stContext
  - defined on, 644
- edFieldsOf:
  - defined on, 591
  - used on, 627, 629, 634
- gValueOf:
  - defined on, 578
  - used on, 580, 620
- AllObjectsAccessibleFrom:suchThat:do:
  - defined on, 678
  - used on, 677, 683
- AllOtherObjectsAccessibleFrom:suchThat:do:
  - defined on, 678, 680
  - used on, 678
- gBitOf:
  - defined on, 662
  - used on, 671, 673, 684, 688
- sh:
  - defined on, 587
- sObject:
  - used on, 636
- lderExtensionOf:
  - defined on, 580
- lderOf:
  - defined on, 577
- idOfFreeChunkList:inSegment:
  - defined on, 666
  - used on, 666-667, 669, 671
- dOfFreePointerList
  - defined on, 665
  - used on, 666
- heapChunkOf:byte:
  - defined on, 663
  - used on, 687
- heapChunkOf:word:
  - defined on, 663
  - used on, 663, 668, 678, 680-681, 684-686
- highByteOf:
  - defined on, 575
  - used on, 617
- homeContext, 584
  - defined on, 583
  - used on, 583, 600
- increaseReferencesTo:
  - defined on, 571, 687
  - used on, 585, 610
- initialInstanceOf:
  - defined on, 573, 688
  - used on, 637
- initialInstructionPointerOfMethod:
  - defined on, 578
  - used on, 606
- initializeAssociationIndex
  - defined on, 599
- initializeClassIndices
  - defined on, 587
- initializeContextIndices
  - defined on, 581
- initializeGuaranteedPointers
  - defined on, 576
- initializeMessageIndices
  - defined on, 590
- initializeMethodCache
  - used on, 605, 647
- initializeMethodIndices
  - defined on, 577
- initializePointIndices
  - defined on, 625
- initializeSchedulerIndices
  - defined on, 641
- initializeSmallIntegers
  - defined on, 575-576
- initializeStreamIndices
  - defined on, 631
- initPrimitive
  - defined on, 616
  - used on, 618, 620
- instanceAfter:
  - defined on, 573, 688
  - used on, 637
- instancesOf:
  - used on, 637
- instanceSpecificationOf:
  - defined on, 590
  - used on, 591

methodClassOf:  
   defined on, 580  
   used on, 607  
 newActiveContext:, 610  
   defined on, 585  
   used on, 606, 609, 639-640, 643  
 newMethod, 589  
   defined on, 587  
   used on, 588, 605, 620, 640-641  
 newProcess  
   defined on, 642  
   used on, 643  
 newProcessWaiting  
   defined on, 642  
   used on, 643-644  
 nilContextFields  
   defined on, 610  
   used on, 610  
 objectPointerCountOf:  
   defined on, 578  
   used on, 633-634  
 obtainPointer:location:  
   defined on, 670  
   used on, 670, 674  
 oddBitsOf:  
   defined on, 662  
   used on, 685, 688  
 ot:  
   defined on, 661-662  
   used on, 670  
 ot:bits:to:  
   defined on, 662  
   used on, 662  
 pointerBitOf:  
   defined on, 662  
   used on, 685-686, 688  
 pop:, 597  
   defined on, 585  
   used on, 590, 606, 639-640  
 popInteger  
   defined on, 617  
   used on, 622-624, 633-637  
 popStack, 597  
   defined on, 585  
   used on, 600-602, 609, 617, 620, 625,  
   628-639, 641, 647, 653  
 popStackBytecode  
   defined on, 601  
   used on, 598, 600  
 positive16BitIntegerFor:  
   defined on, 617  
   used on, 628-629

positive16BitValueOf:  
   defined on, 617-618  
   used on, 628-630, 634  
 primitiveAdd  
   defined on, 562 622  
 primitiveAsObject  
   defined on, 636  
   used on, 633  
 primitiveAsOop  
   defined on, 636  
   used on, 633  
 primitiveAt  
   defined on, 628  
   used on, 627  
 primitiveAtEnd, 631  
   defined on, 632  
   used on, 627  
 primitiveAtPut, 628  
   used on, 627  
 primitiveBecome  
   defined on, 635  
   used on, 633  
 primitiveBitAnd  
   defined on, 624  
   used on, 619, 622  
 primitiveBitShift  
   defined on, 624  
   used on, 619, 622  
 primitiveBlockCopy  
   defined on, 638  
   used on, 637  
 primitiveClass  
   defined on, 653  
   used on, 619, 652  
 primitiveDiv  
   defined on, 623  
 primitiveDivide  
   defined on, 622  
   used on, 619, 622  
 primitiveEqual  
   defined on, 624  
   used on, 619, 622  
 primitiveEquivalent  
   defined on, 653  
   used on, 619, 652  
 primitiveFail  
   defined on, 616  
   used on, 574, 617-619, 625, 637  
 primitiveFlushCache  
   defined on, 647  
   used on, 638  
 primitiveIndex  
   defined on, 587  
   used on, 588, 605, 620



## Part 5: Summary of System Interface Components

In this section, each main kind of menu and cursor, and each kind of view in the basic Smalltalk-80 system is summarized. Fuller description of these interface components is contained in *Smalltalk-80: The Interactive Programming Environment*. This section is intended to summarize that information. The summary of the views consists of two or three diagrams of the subviews of each view and a description of the various aspects of the view. The three possible diagrams are:

<b>Descriptions</b>	Indicates the kind of information that appears in the subview. If TEXT is indicated, then the subview is a standard text editor. If LIST MENU is indicated, then the subview contains a fixed list menu.
<b>Yellow-Button Menus</b>	Displays the pop-up menu that appears if the yellow button is pressed while the cursor is in the subview. (The order of the menu items might not be what you see on your display screen; the ordering is easy to change and is a place where people's personal preferences show up. There may also be additional items.)
<b>Dependencies</b>	Arrows connecting the subviews indicate that actions carried out in one subview (tail of the arrow) affect the information displayed in another subview (head of the arrow).

The various aspects of the views that are covered in the textual descriptions include:

<b>how accessed</b>	The circumstances under which the view appears on the screen.
<b>how created</b>	What, if anything, you must do to assist in displaying the view.
<b>how terminated</b>	What you do to remove the view from the screen.
<b>blue button activity</b>	What items appear in the pop-up menu when you press the blue button. "Default" indicates that the standard system view blue button appears.
<b>red button activity</b>	What selection actions you can take by pressing the red button.
<b>yellow button activity</b>	What items appear in the pop-up menu when you press the yellow button.

Each view may contain several subviews. These are labeled A, B, C, and so on. The red, yellow, and blue button activities may differ in each subview.

accessible views	Which other views can be created as a result of actions in this view. Only those accessible by selecting menu items are listed. Other views can appear; for example, errors in evaluation create notifiers.
keyboard activity	What keys on the keyboard can affect the view.
message-sending	Ways of affecting the view by sending messages to it. Assumes there is a variable referencing the view that is available to the user.

## 5.1—System Menus, Cursors, and Text Editor

### 5.1.1 System Menu

### 5.1.2 Standard System View Blue Button Menu

### 5.1.3 ScrollBars

### 5.1.4 Confirmers

### 5.1.5 Prompters

### 5.1.6 Cursors

### 5.1.7 Text Editing

Text Selection

Text Editing Yellow Button Menu

Issuing Commands Using Keys and the "Control" Key

Inserting Delimiters About a Selection

Menus provide an interface for obtaining information in the Smalltalk-80 system. Each view contains information, much of which can be edited. The yellow button is used to bring up a menu of messages that can be sent to the view in order to edit its contents. The blue button is used to bring up a menu of messages that edit the view itself; these are common to views throughout the system.

### 5.1.1 System Menu

The menu obtained by moving the cursor into the light gray background area and then pressing the yellow button is called the *System Menu*. The menu is shown in Figure 5.1.1.

restore display	Redraws the display, getting rid of anything which is not known to the control manager. Another effect of restoring the display is to reset the cursor to the slanted arrow (normal) cursor.
exit project	It is possible to create several different collections of views of information, each one called a <i>project</i> . Each project takes up an entire display screen for the presentation of its views. A project is accessed by creating a project view and then choosing the yellow button menu command <i>enter</i> . Once inside a project, it is possible to return to the project from which it was entered by choosing this command. At the topmost project, this command is equivalent to <i>restore display</i> .
project	Creates a new project view. You are asked to designate a rectangular area in which the project view is to be displayed.
file list	Creates a view of the local files. You are asked to designate a

	rectangular area in which a file list view is to be displayed.
browser	Creates a new system browser that allows you to access hierarchically-organized information about the Smalltalk-80 system itself. You are asked to designate a rectangular area in which the browser view is to be displayed.
workspace	Creates a blank area in which to edit text. You are asked to designate a rectangular area in which the workspace is to be displayed.
system transcript	Creates a view of the System Transcript. You are asked to designate a rectangular area in which the transcript is to be displayed.
system workspace	Creates a view of the System Workspace, any useful message expressions that you can edit and evaluate, notably expressions about accessing files, querying the system, and recovering from a crash. You are asked to designate a rectangular area in which the workspace is to be displayed.
save	Stores the complete image of the system in an external file.
quit	When you are done working, select this command.

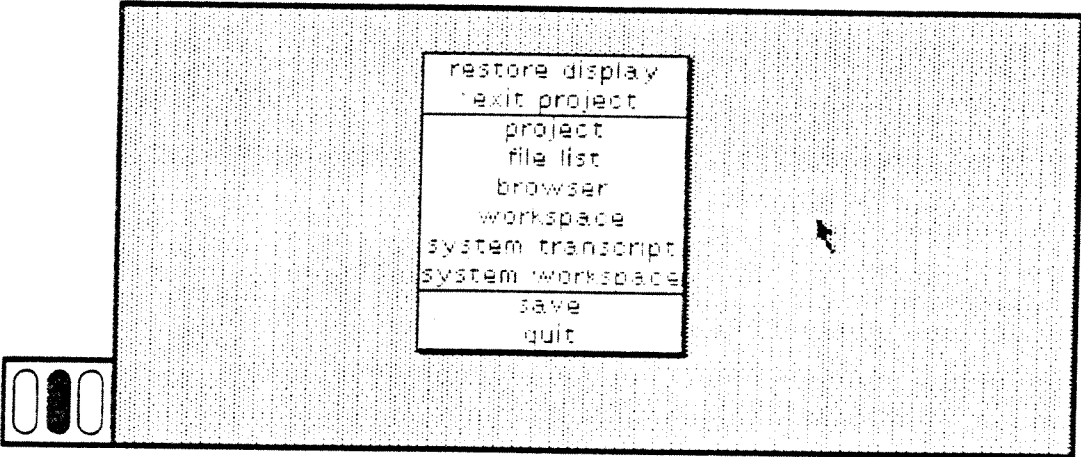


Figure 5.1.1. System Menu

5.1.2 Standard System View Blue Button Menu

When you press blue button while the cursor is inside a standard system view, a menu for modifying the view appears. It looks as shown in the workspace in Figure 5.1.2.

under	Selects a view (if one exists) that is both under the active view and under the cursor.
move	Designates a new origin for the view. The view disappears with only its label remaining, and the cursor changes to the shape of the origin cursor. Move the cursor around. The label of the view tracks the cursor. When you press the red button, the view reappears so that the label is at the last location to which you moved it.
frame	Designates a new rectangular area for the view.
collapse	Replaces the view with an area containing its label only.
close	Erases the view from the display screen and deletes the view from the system.

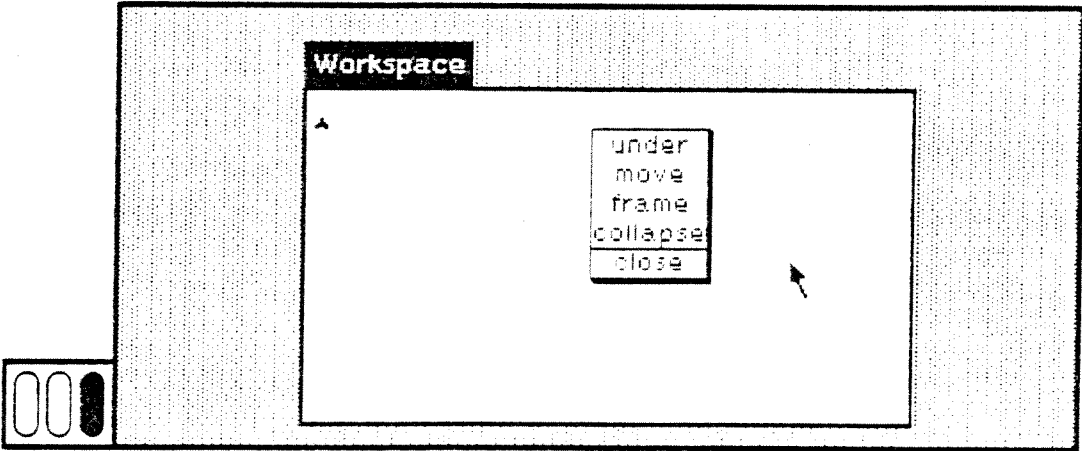


Figure 5.1.2. Standard System View Blue Button Menu

### 5.1.3 ScrollBars

A view on the screen may not be large enough to display all the information appropriate to that view. Assume you wish to examine a large document of information, and that you have created a view on the screen for this purpose: a *view* (or often called a *window*) is defined by mapping from the area allotted on the display screen to a corresponding-sized area of the document. The mapping determines how much document information can actually be displayed in the screen view. The purpose of a scroll bar is to change the area of the document that can be seen in the view. It is done in one of three ways.



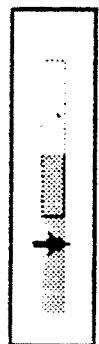
#### scroll next

The line of text nearest the cursor is moved to the top of the view. Move the cursor into the scroll bar area, in the right one-third of the rectangle and outside of the gray area. The cursor shape becomes that of an up arrow. Click the red button. Scrolling occurs.



#### scroll previous

The line of text at the top of the view moves to become the line nearest to the cursor. Move the cursor into the scroll bar area, in the left one-third of the rectangle and outside of the gray area. The cursor shape becomes that of a down arrow. Click the red button. Scrolling occurs.



#### jump

Displays a view of the document beginning with a location in the document relative to the gray area in the scroll bar. Move the cursor into the scroll bar area, in the middle one-third of the rectangle and outside of the gray area. The cursor shape becomes that of a right arrow. Press the red button and hold. The gray area moves to the cursor location and then tracks the cursor until the red button is released. The displayed document jumps to the appropriate location. While the red button remains pressed, a lighter gray image is left in the scroll bar area to indicate the previous position of the gray area.

### 5.1.4 Confirmers

A *confirmer* is a "binary choice" menu. That is, a confirmer is a menu with two items in it, where each choice represents opposing points of view. A confirmer consists of three parts, the top part is a statement or question; the other two parts are possible opinions about the

statement or answers to the question. When the cursor moves over the two options, the shape of the cursor changes to either a hand with the thumb pointing upward, or a hand with the thumb pointing downward.

There are several places in the system where confirmers appear. In the system class browsers, confirmers appear when you try to

- remove a class category
- remove a class
- remove a method from a class
- remove a message protocol

If the spelling corrector is invoked, in a workspace, in the code part of a system browser, or in a notifier or debugger, it may propose a correction that will require your confirmation. Whenever you try to close a view whose content has not been saved, a confirmer will appear. If you try to replace the contents of a view of text when the existing contents have not been saved, a confirmer will appear to make certain that you want to discard text changes. And when you try to delete a file listed in a file list view or you try to remove a dictionary entry listed in a Dictionary inspector, a confirmer will appear to make certain that you really want to proceed with the deletion.

Confirmers are instances of class `BinaryChoice`. You can browse all references to the class to see examples of its use.

### 5.1.5 Prompters

A *prompter* is a "fill in the blank" menu. That is, a prompter is a menu in which you must type your choice. It consists of two parts: the top part is a statement or question, and the bottom part is a workspace in which you type.














There are several places in the system in which a prompter appears. In a system browser, a prompter appears when you

- add a class category
- add message protocol to a class
- move a message from one protocol to another
- rename a class category
- rename a class
- rename a message protocol

A prompter appears in the change management browser when you name files for reading or writing changes files, and when you name a snapshot file. In the Form Editor, a prompter is used for file naming, and for setting the size of the graphical grids. Dictionary inspectors use a prompter to request the name of a key for a new entry.

Prompters are instances of class `FillInTheBlank`. You can see examples of the use of prompters by browsing all references to the class.

### 5.1.6 Cursors

<u>name</u>	<u>image</u>	<u>use</u>
normal		The cursor looks like this most of the time. The point of cursor selection is at the upper left corner, at the tip of the arrowhead.
execute		Wait. The system is executing some expression. During this time, you cannot do anything else.
origin		Indicates that you should designate the top left corner of a rectangular area by moving the cursor to where you want, and then pressing, <i>but not releasing</i> , the red button.
corner		Indicates that you should designate the bottom right corner of a rectangular area by keeping the red button depressed while you move the cursor to where you want the corner to be, then releasing the button.
read		Wait. Information is being read from an external file.
write		Wait. Information is being written on an external file.
crossHair		In the Bit Editor, indicates the location of the bit at which editing will occur.
down		(previous information) In a scroll bar, indicates scrolling the text to see the preceding text.
up		(next information) In a scroll bar, indicates scrolling the text to see the succeeding text.
marker		(jump) In a scroll bar, indicates the proportional location to which you want to jump.
wait		Wait. The system is carrying out some file operation that is time consuming.
thumbs up		Answer yes in the confirmer.
thumbs down		Answer no in the confirmer.

### 5.1.7 Text Editing

Text editing is carried out using a combination of commands issued by selecting an item from

the yellow button menu and commands issued by typing keys on the keyboard.

## Text Selection

Text is selected using the pointing device and the red button. Move the cursor in a view of text, either at one of the characters or between characters or at the end of the passage of text. Click the red button. A caret (an inverted "v") appears at the cursor location or at the gap just before the character. Pointing to a place in the text and clicking the red button creates a zero-width selection. The method of clicking once between characters is the one to use if you want to insert text.

Move the cursor to one end of the passage of text and press the red button. Hold it down while moving the cursor to the other end of the passage. This activity is called *draw through*. When the cursor reaches the other end of the passage, release the button. The selected text is highlighted. The method of drawing through a passage of text is the one to use if you want to replace, copy, delete, or change the font or emphasis (bold or italic face) of the text.

Clicking the button twice with the cursor in the same location selects different passages, depending on the cursor location.

<u>To select</u>	<u>Double click</u>
a whole word	within a word, or just before or just after the word if the word is not just inside a delimiter
a delimited text	just after the left member of a pair of delimiters or just before the right member (the delimiters themselves are not selected); recognized delimiters are parentheses, square brackets, angle brackets, braces ("curly brackets"), single quotes, and double quotes
all text in the view	at the beginning or the end of everything in the view
a line of text	(if the line is delimited by carriage returns) at the beginning of a line (just after a carriage return), or at the end of a line (just before a carriage return)

You can also select text that is not visible in the view by drawing through to the outside of the view, causing automatic scrolling to the next text (if you move outside the bottom of the view) or scrolling to the previous text (if you move outside the top of the view).

The "escape" key on the keyboard can be used to select text that was just typed. Press the "escape" key when you have finished typing. The characters you typed since the last mouse click will be highlighted.

## Text Editing Yellow Button Menu

When you press the yellow button while the cursor is in a view of editable text, messages pertaining to modifying that text appear on a menu (see Figure 5.1.3).

again	Repeats the last replace, copy, or cut edit command.
undo	If possible, reverses the effect of the last command.
copy	Places a copy of the current text selection into a buffer.
cut	Deletes the current text selection; saves it in a buffer.
paste	Text is remembered in a buffer when copied or cut. Replaces the current text selection with the remembered text.

Messages may appear that pertain to evaluating the text.

do it	Evaluates the text as a Smalltalk-80 expression. This involves first compiling the expression to check for syntax errors.
print it	Same as do it except a description of the result of evaluation is inserted into the text and becomes the current selection.

Messages may also appear that pertain to storing the text or to retrieving a copy of the text before any editing was done to it and after the last time the accept command was issued.

accept	Stores the text. This command has a variety of meanings, depending on the context in which the text was created. In a workspace, the text is simply remembered so that if more editing is done and then cancel is chosen, the stored text can be restored in the workspace. In a browser, the text is compiled. If the compilation is successful, the compiled method is stored.
cancel	Restores the text in the view to the version at the last time an accept was issued; if no accept was issued, then the text is restored to the version when the text was first displayed in the view.

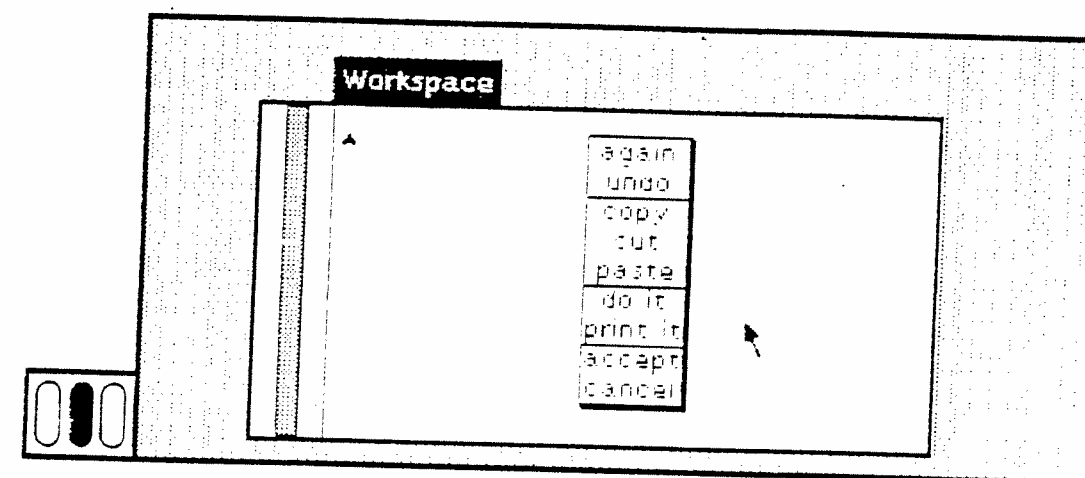


Figure 5.1.3. Text Editing Yellow Button Menu

Issuing Commands Using Keys and the "Control" Key

<u>type keys</u>	<u>action</u>
ctrl 0	Change the font of the text selection. The system default is sans-serif 12-point font. Changing a text selection to bold, italic, or underline is considered a change in font and is done by selecting the appropriate font number. The numbers depend on the current style being used.
ctrl 1	Change the font of the text selection. The system default is sans-serif 10-point font.
ctrl 2	Change the font of the text selection. The system default is sans-serif 10-point font, bold.
ctrl 3	Change the font of the text selection. The system default is sans-serif 10-point font, italic.
ctrl 4	Change the font of the text selection. The system default is serif 12-point font.
ctrl 5	Change the font of the text selection. The system default is serif 12-point font, bold.
ctrl 6	Change the font of the text selection. The system default is serif 12-point font, italic.
ctrl 7	Change the font of the text selection. The system default is serif 10-point font.
ctrl 8	Change the font of the text selection. The system default is serif 10-point font, bold.
ctrl 9	Change the font of the text selection. The system default is serif 10-point font, italic.
ctrl t	insert the text ifTrue:.
ctrl f	insert the text ifFalse:.
ctrl w	cut the text selection and the word preceding the caret (typically used while typing and while there is no text selection simply to delete the last word typed).
"delete"	cut the current text selection.
"backspace"	cut the text selection and the character before the selection.
ctrl -	Underline the selected text.
ctrl b	Make the selected text boldfaced.

ctrl c	This is the system interrupt and should not be typed while text editing unless a process interrupt is desired.
ctrl shift -	Remove any underline from the selected text.
ctrl shift b	Make the selected text not boldfaced.
ctrl shift c	This is a special system interrupt, used when the system seems dead and everything else fails.

Inserting and Deleting Delimiters About a Selection

If you type one of the following, it will either insert or remove the corresponding delimiter about the selection. If the selection is currently between the corresponding delimiters, then the delimiters will be deleted. Otherwise, they will be inserted.

<u>type keys</u>	<u>insert delimiters</u>
ctrl [	[ and ]
ctrl (	( and )
ctrl <	< and >
ctrl "	" and "
ctrl '	' and '



5.2—System Views

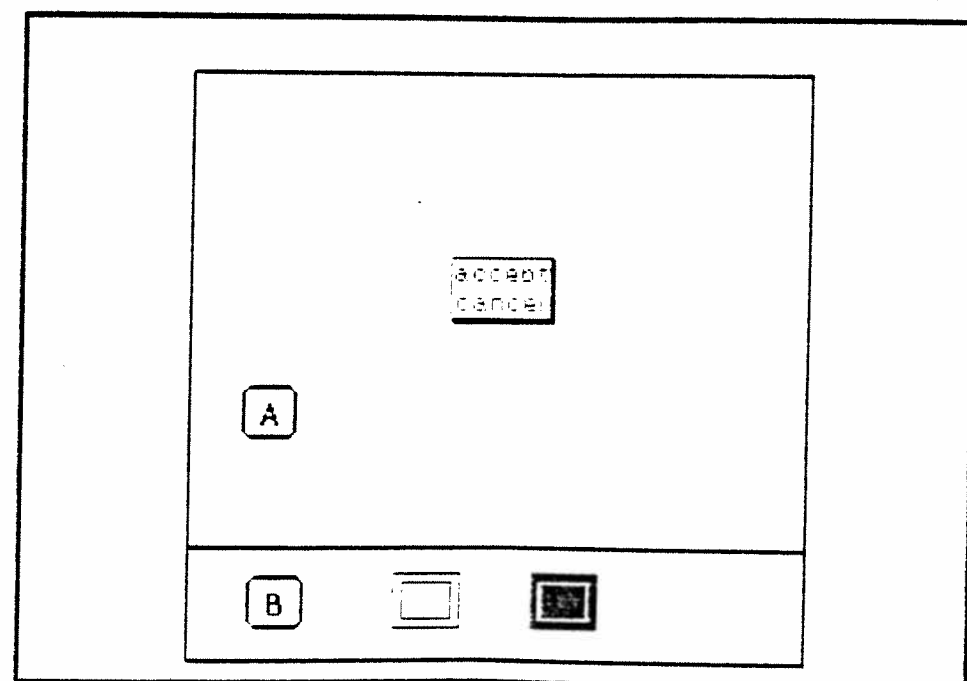
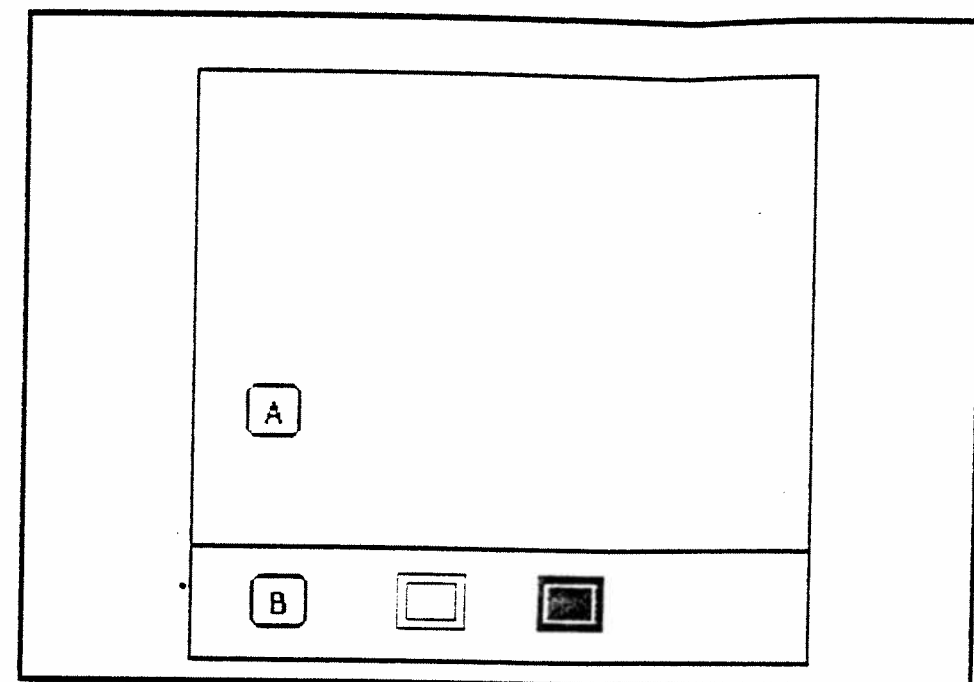
- 5.2.1 Bit Editor
- 5.2.2 System Class Browser
- 5.2.3 Change-Management Browser
- 5.2.4 Change-Set Browser
- 5.2.5 Debugger
- 5.2.6 File List
- 5.2.7 Form Editor
- 5.2.8 Inspector
- 5.2.9 Message-Set Browser
- 5.2.10 Notifier
- 5.2.11 Project
- 5.2.12 Syntax Error
- 5.2.13 Text Collector
- 5.2.14 Workspace

The system views outlined in this section are presented in alphabetical order.

5.2.1 Bit Editor

There are two ways to access the Bit Editor, one that creates a standard system view in which a Form is edited using the Bit Editor (this is a "scheduled" Bit Editor), and one that is not embedded in a standard system view (this is an "unscheduled" Bit Editor).

name	Bit Editor (scheduled)
general description	Used to create and modify a form by placing black or white dots in a magnified view of the form. The cursor shape is the crossHair cursor.
how accessed	<p>To create a view with two subviews, a magnified one in which editing is one and an unscaled version, evaluate one of the following expressions (where extentPoint and originPoint are instances of class Point; scaleFactor is a Point; aForm1 and aForm2 are instances of class Form, both of which must be the same size).</p> <p>Form fromUser bitEdit</p> <p>(Form new extent: extentPoint) bitEdit</p> <p>(Form new extent: extentPoint) bitEditAt: originPoint</p>



Form fromUser bitEditAt: originPoint scale: scaleFactor

(OpaqueForm shape: aForm) bitEdit

(OpaqueForm figure: aForm1 shape: aForm2) bitEdit

Try as an example OpaqueForm makeStar bitEdit.

**how created** Designate the top left corner of the rectangular area. The extent of the area is fixed by the size of the Form. In the case of the first expression, you designate the Form's rectangular area first.

**how terminated** Blue button choose close.

**blue button activity** Default

**red button activity** In subview A, place a color dot at the location of the cursor. Any change to the image is reflected in the view in subview C. Choose a color from the menu in subview B.

**yellow button activity**

subview A

**accept** Save the current displayed image as the form.  
**cancel** Restore the saved form as the current displayed image.

**keyboard activity** Each "color" menu item corresponds to a key on the keyboard. Pressing the key is identical to selecting the menu item with the red button. n=black, v=gray (when available), x=white.

**name** Bit Editor (unscheduled)

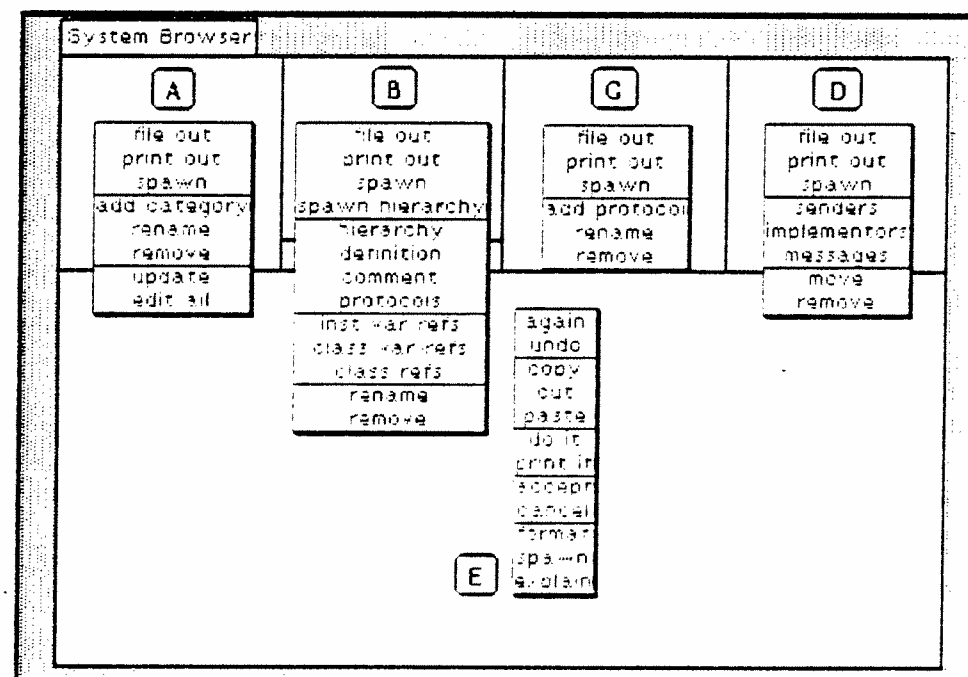
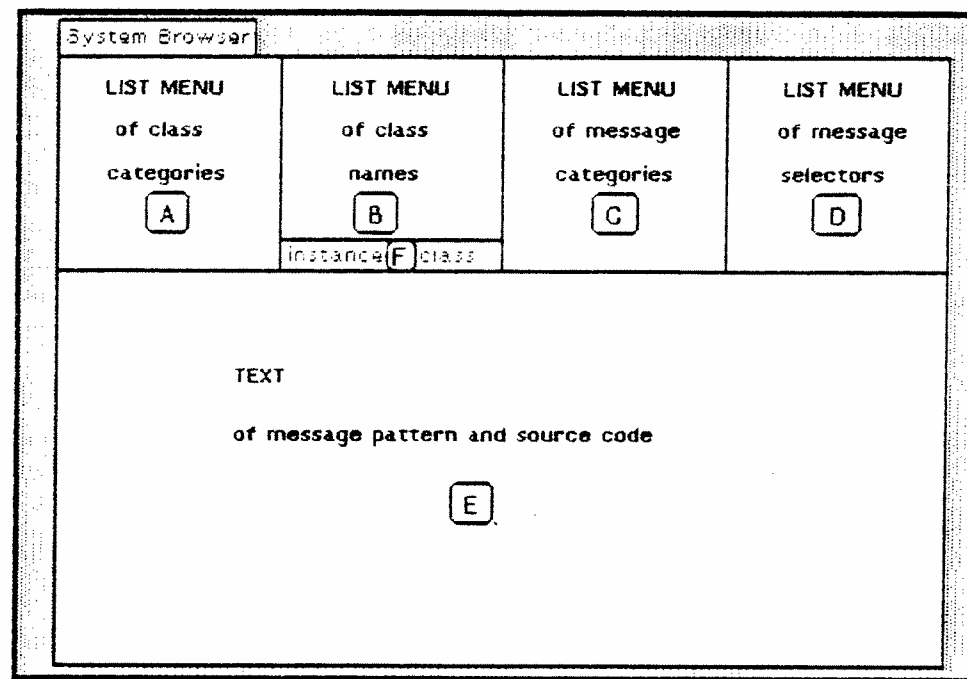
**general description** Used to create and modify forms by placing black or white dots in a magnified view of a form. The cursor shape is the crossHair cursor.

**how accessed** Evaluate an expression of the form

BitEditor magnifyOnScreen

**how created** Designate the Form's rectangular area and then designate the top left corner of the rectangular area for the magnified view. Extent of the area is fixed as a scale of the size of the Form's area.

**how terminated** Click any button outside the editing area.



blue button activity None

red button activity Place a "color" dot at the location of the cursor. Choose a color from the menu in subview B.

yellow button activity

subview A

accept

Save the currently displayed image as the form.

cancel

Restore the saved form as the current displayed image.

keyboard activity Each "color" menu item corresponds to a key on the keyboard. Pressing the key is identical to selecting the menu item with the red button. n=black, v=gray (when available), x=white.

## 5.2.2 System Class Browser

The use of only the full system class browser is summarized here. The other browsers are subsets of this full browser, where the "red button activities," "yellow button menus," and "how accessed" and "how terminated" are all the same for those subviews shared in common. These other browsers are the System Category Browser, Class Browser, Message Category Browser, Message Browser, and Class Hierarchy Browser. Typically, these browsers are accessed by choosing yellow button commands.

name

System Browser

general description Used to access system class descriptions for the purpose of reading and editing methods, and to create new class definitions and categories of class definitions. Typically all program development is done in a System Browser or in one of its sub-browsers, such as a Class Browser.

how accessed

System Menu choose browser.

Create a Class Browser by evaluating an expression of the form

Browser newOnClass: <className>

how created

Designate the rectangular area.

how terminated

Blue button choose close.

blue button activity Default

red button activity The menu items in subview F, instance and class, select viewing one of two aspects of a class: messages to instances, or messages to the class itself.

Choose items in the list menus in subviews A, B, C, and D; and select characters for text editing in subview E.

subview A When there is a selection, the names of classes in the selected category display in subview B.

subview B When there is no selection, the class definition template displays in subview E. The template is

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

Edit by replacing each argument--NameOfSuperclass, NameOfClass, instVarName, ClassVarName, Category-Name--with an appropriate name. Class names and class variable names must begin with a capital letter. The message subclass: can be replaced with variableSubclass:, variableByteSubclass:, or variableWordSubclass: if appropriate. These alternative keywords indicate special class representations for classes whose instances have indexable instance variables, classes whose instances have indexable instance variables that are represented as bytes, and classes whose instances have indexable instance variables that are represented as words,

When there is a selection, the names of the message protocols of the selected class display in subview C, and the class definition of the selected class displays in subview E.

See Part 3 for the use of multiple inheritance class definitions.

subview C When there is a selection, the names of message protocols in the selected class display in subview D.

subview D When there is no selection in this subview, the method definition template displays in subview E. The template is

```
message selector and argument names
"comment stating purpose of message"

| temporary variable names |
statements
```

Edit by substituting for all parts to create a well-formed Smalltalk-80 method.

When there is a selection in this subview, the method associated with the selected message selector is displayed in subview E.

subview E Text editing.

yellow button activity

subview A (no item selected)

add category	Adds a new item to the class category menu.
update	Informs this browser if new categories were created by reading from an external file or by editing in another browser.
edit all	Displays in subview E the system class organization as a sequence of class category descriptions, each in the format  ('category' className className)

subview A (item selected)

file out	Prints descriptions of each class in this category onto a file whose name is the category name concatenated with the extension '.st'.
print out	Prints a "pretty printed" description of each class in this category onto a file whose name is the category name concatenated with a system-specific extension.
spawn	Opens a System Category Browser in which only the classes included in the selected category can be accessed.
add category	Adds a new item to the class category menu.
rename	Changes the name of the currently selected category.
remove	Removes this category of classes from the system.
update	Informs this browser if new categories were created by reading from an external file or by editing in another browser.
edit all	Displays in subview E the system class organization as a sequence of class category descriptions, each in the format  ('category' className className)

subview B (no item selected)

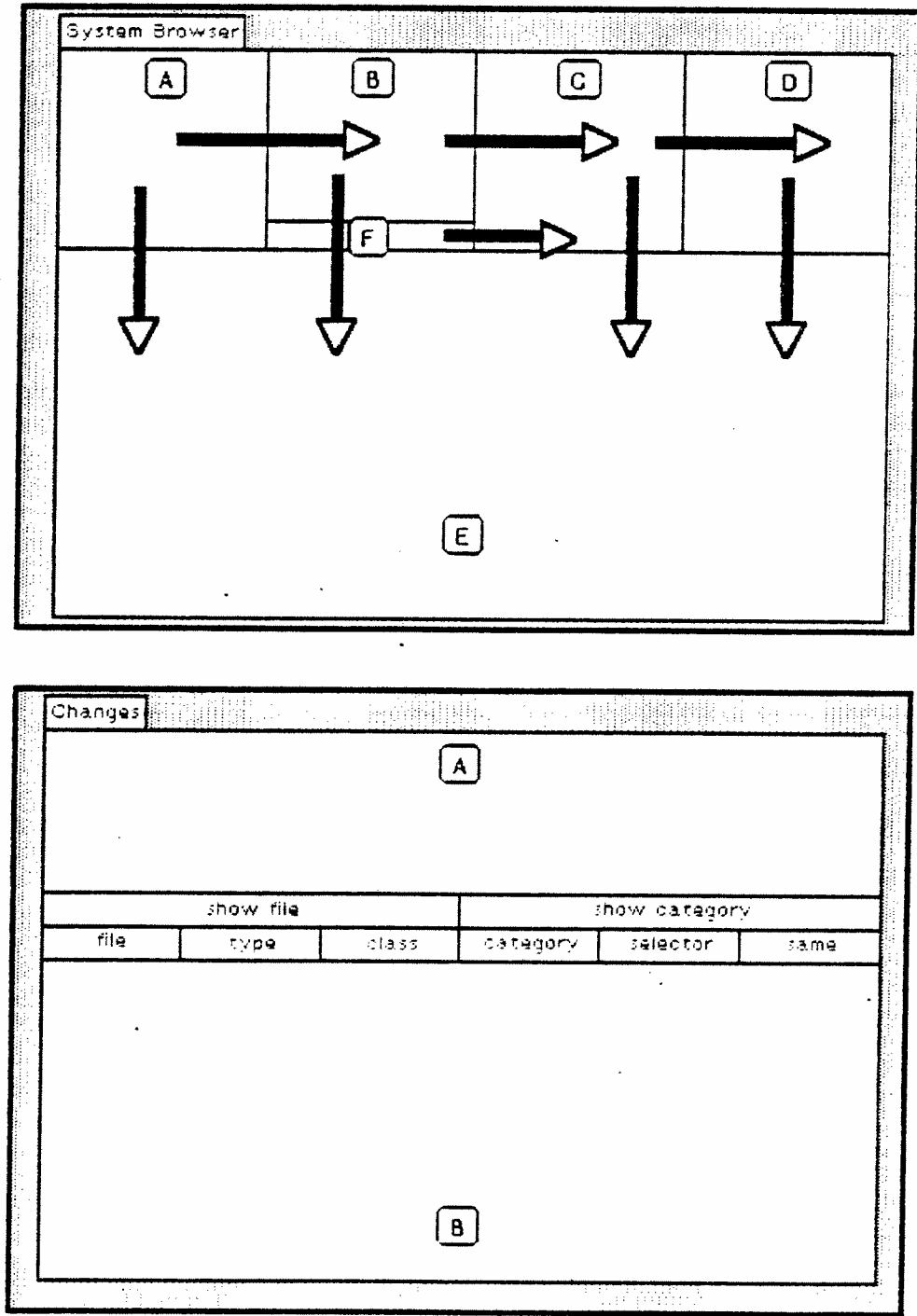
No menu displays. Subview flashes.

subview B (item selected)

file out	Prints the description of the selected class onto a file whose name is the class name concatenated with the extension '.st'.
print out	Prints a "pretty printed" description of the selected class onto a file whose name is the class name

	concatenated with a system-specific extension.
spawn	Opens a Class Browser in which only the description of the selected class can be accessed.
spawn hierarchy	Opens a Class Hierarchy Browser in which only the descriptions of the selected class, its superclasses, and its subclasses, can be accessed.
hierarchy	Displays in subview E a description of the class hierarchy of the selected class. The description includes the instance variables of each class.
definition	Displays in subview E the message that creates the selected class.
comment	Displays in subview E the comment describing the purpose of the selected class.
protocols	Displays in subview E the class message organization as a sequence of message category descriptions, each in the format ( <code>'category'</code> <code>messageSelector</code> <code>messageSelector</code> )
inst var refs	Displays a menu of the instance variables of instances of the selected class. Choose one to open a Message-Set Browser for all methods that refer to the selected instance variable.
class var refs	Displays a menu of the class variables of the selected class. Choose one to open a Message-Set Browser for all methods that refer to the selected class variable.
class refs	Opens a Message-Set Browser for all methods that refer to the selected class.
rename	Changes the name of the selected class and opens a Message-Set Browser for all methods that refer to the class.
remove	Removes the selected class from the system.
subview C (no item selected)	
add protocol	Adds another message category to the protocol menu.
subview C (item selected)	
file out	Prints the description of the selected class's selected protocol onto a file whose name is the class name followed by a hyphen followed by the protocol and concatenated with the extension <code>'st'</code> .
print out	Prints a "pretty printed" description of the selected class' selected protocol onto a file whose name is the class name followed by a hyphen followed by the protocol and concatenated with a system-specific

	extension.
spawn	Opens a Message Category Browser on the selected protocol in which only the description of the selected protocol of messages can be accessed.
add protocol	Adds another message category to the protocol menu.
rename	Changes the name of the currently selected protocol.
remove	Removes the selected protocol from the class.
subview D (no item selected)	No menu displays. Subview flashes.
subview D (item selected)	
file out	Prints the description of the selected message onto a file whose name is the class name followed by a hyphen followed by the selector and concatenated with the extension <code>'st'</code> .
print out	Prints a "pretty printed" description of the selected message onto a file whose name is the class name followed by a hyphen followed by the selector and concatenated with a system-specific extension.
spawn	Opens a Message Browser in which only the description of the selected message for the selected class can be accessed.
senders	Opens a Message-Set Browser to access all the methods in which the selected message is sent.
implementors	Opens a Message-Set Browser to access all the methods that implement the selected message.
messages	Creates a menu of all the messages sent in the method associated with the selected message. Choose one to open a Message-Set Browser to access all the methods that implement the selected message.
move	Moves the selected message to another protocol.
remove	Removes the selected message from the selected class.
subview E	
again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.



- print it as in evaluating expressions.
- accept If the text is from hierarchy, do nothing.  
If the text is from edit all or protocols, store the new categorization.  
If the text is a definition or comment, evaluate the text as a message. This evaluation might alter the definition or the class comment. Generally the text is evaluated as a message and might fail if variables are undeclared.  
In the previous cases we assumed that no message protocol was selected. If a protocol is selected, then the text is a method that is compiled and stored in the method dictionary of the selected class under the selected protocol. If this message is already stored under a different protocol, it is removed from that protocol and stored under the currently selected one.
- cancel Redisplays the text that appeared before any editing that occurred after the last accept.
- format For methods only, redisplays the text, pretty-printed. Does not do an automatic accept since you might not like the formatted version. Does not work if the text has been edited since the last accept.
- spawn Creates a Message Browser for the selected message in which the method is the edited (but not yet saved) version currently displayed. Does an automatic cancel in subview E.
- explain Inserts an explanation of the single syntactic element that is the current text selection.

subview dependencies Text is displayed in subview E as the result of selections in A, B, or D. If there is no selection in C, E is the class template for the selection in subview B. The menus displayed in B, C, and D are chosen with respect to the selections in A, B, and C respectively. The selection in F, combined with the selection in B, determines categories in C.

accessible views Class Category Browser, Class Browser, Message Category Browser, Message Browser, Mesage-Set Browser, Class Hierarchy Browser

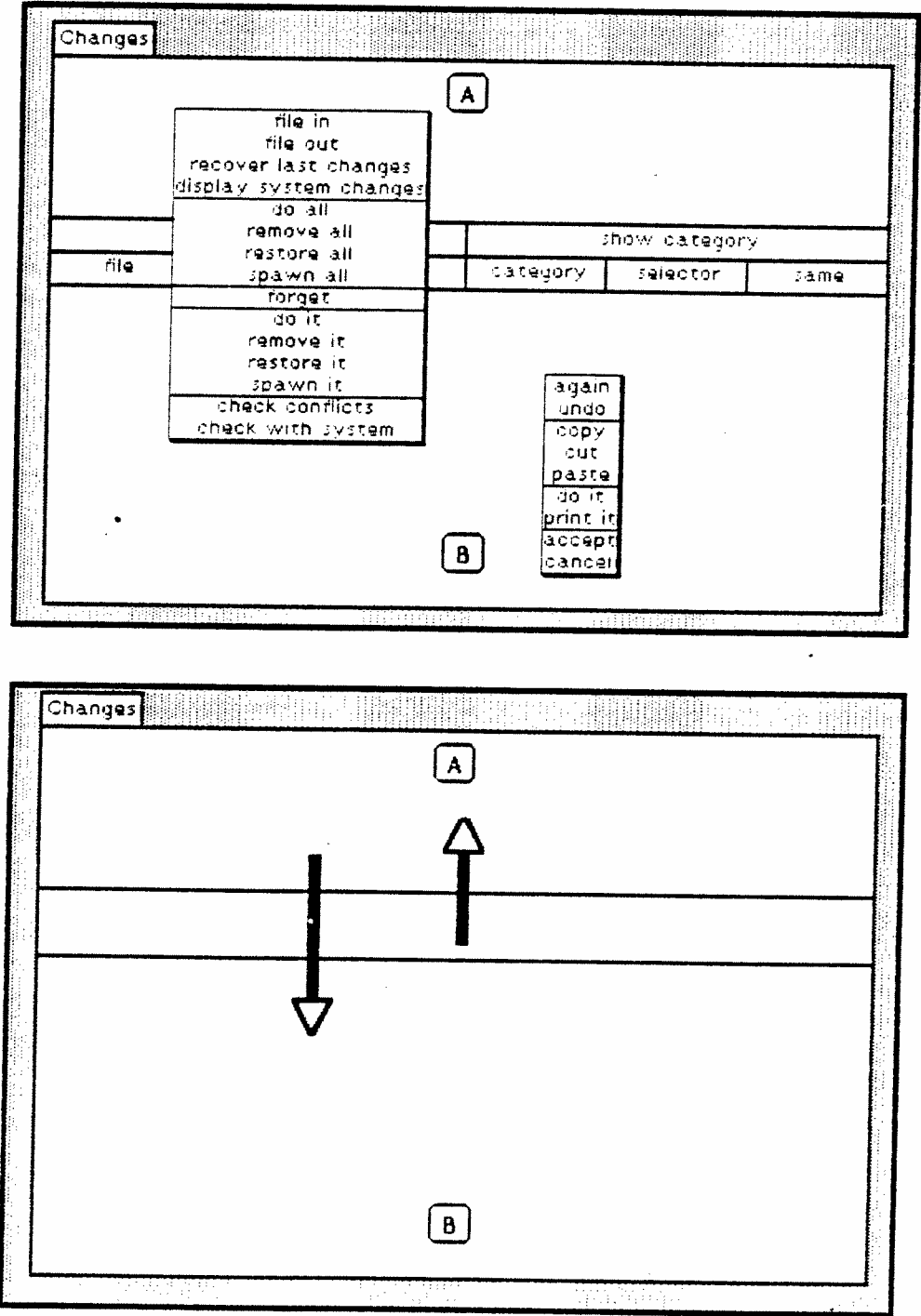
5.2.3 Change-Management Browser

name Change-Management Browser

general description Used to access the changes for crash recovery or combining group work.

how accessed Evaluate one of the following expressions

- ChangeListView open
- ChangeListView recover



ChangeListView  
openOn:  
    (ChangeList new recoverFile:  
      (FileStream oldFileName: 'filename'))

how created      Designate the rectangular area.

how terminated    Blue button choose close

blue button activity Default

red button activity Choose menu items in subview A; choose filters; and select characters in subview B.

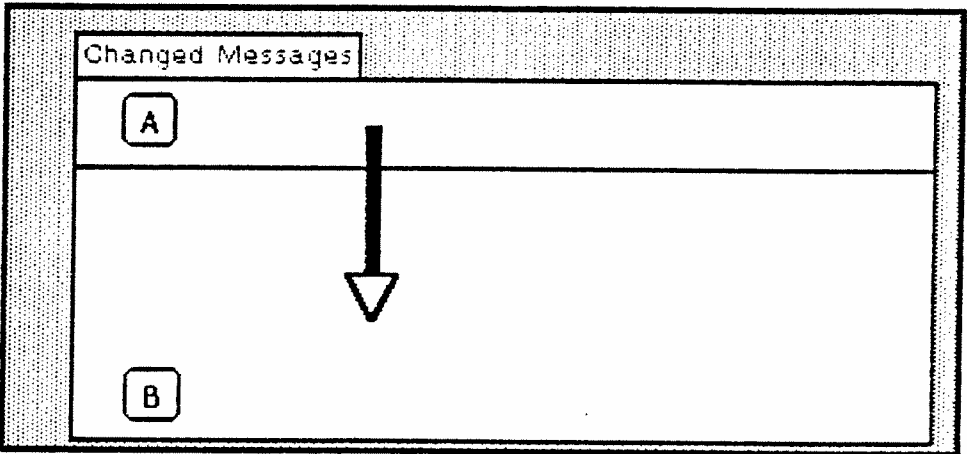
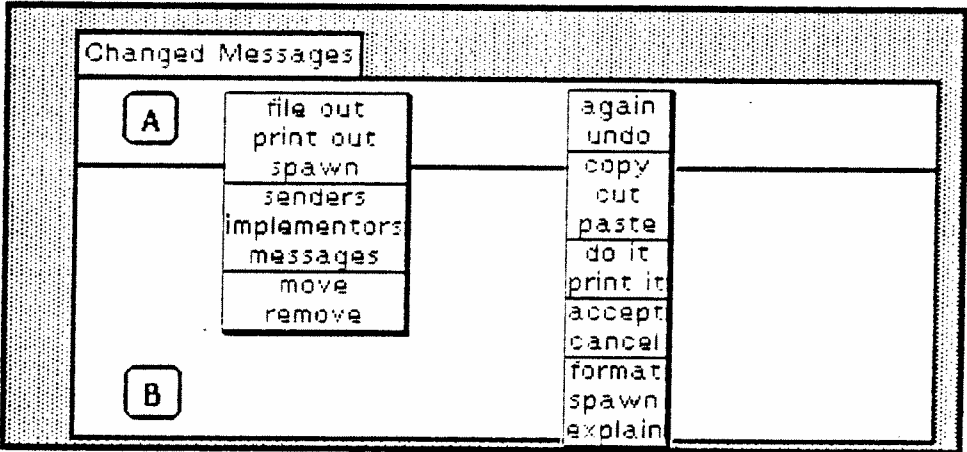
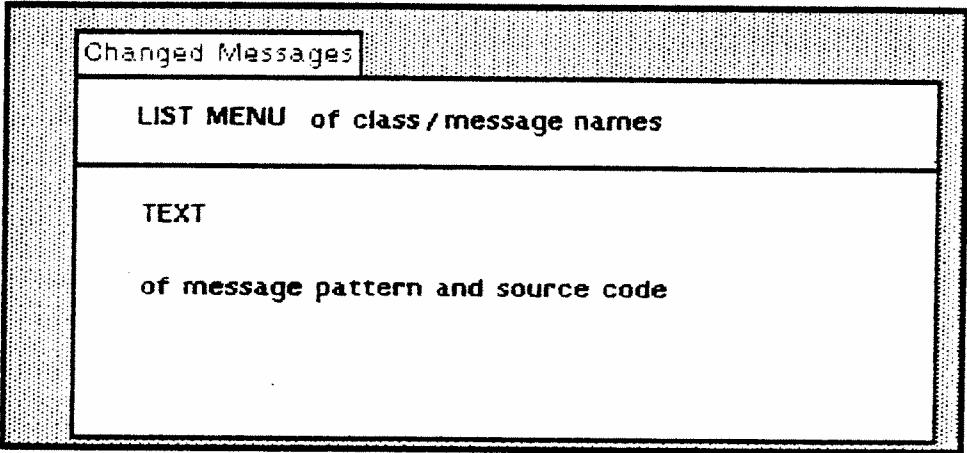
subview A    When there is a selection, the method or expression associated with the selection displays in subview B. The choice of items in subview A depends on the selection of filters.

yellow button activity

subview A

- file in      Adds references to a specified changes file to the list menu.
- file out     Prints all the unmarked items in the list menu onto a new changes file.
- recover last changes    Adds to the list menu, references to the methods in the internal change set.
- display system changes    Adds to the list menu, references to the methods in the internal change set.
- do all      Evaluates each expression or new definition that is referenced in the list menu and is not marked for removal.
- remove all    Marks every item in the list menu for removal.
- restore all    Unmarks every item in the list menu that is currently marked for removal.
- spawn all    Creates another change-management browser whose list menu is identical to the currently displayed one.
- forget      Deletes every item in the list menu that is marked for removal.
- do it      Evaluates the selected item, regardless of whether it is marked for removal.
- remove it    Marks the selected item.





restore it	Unmarks the selected item.
spawn it	Opens a Message Browser for the current system definition of the selected item.
check conflicts	Print any conflicts among the items in the menu onto a file.
check with system	Print any conflicts among the items in the menu and system definitions onto a file.

subview B

again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.
print it	as in evaluating expressions.
accept	Saves the edited text as the actual text.
cancel	Redisplays the text that appeared before any editing that occurred after the last accept.

accessible views Message Browser

5.2.4 Change-Set Browser

name Change-Set Browser

general description Used to access the changes stored in the internal change set.

how accessed Evaluate the following expression

Smalltalk browseChangedMessages

If no changed messages exist, then the characters Nobody appear in the (visible) System Transcript.

how created Designate the rectangular area.

how terminated Blue button choose close.

blue button activity Default

**red button activity** Choose menu items in subview A; select characters in subview B.

subview A When there is a selection, the method associated with the selection displays in subview B.

**yellow button activity**

subview A (no item selected)

No menu displays. Subview flashes.

subview A (item selected)

**file out** Prints the description of the selected message onto a file whose name is the class name followed by a hyphen followed by the message selector and concatenated with the extension '.st'.

**print out** Prints a "pretty printed" description of the selected message onto a file whose name is the class name followed by a hyphen followed by the message selector and concatenated with a system-specific extension.

**spawn** Opens a Message Browser in which only the description of the selected message can be accessed.

**senders** Opens a Message-Set Browser to access all the methods in which the selected message is sent.

**implementors** Opens a Message-Set Browser to access all the methods that implement the selected message.

**messages** Creates a menu of all the messages sent in the method associated with the selected message. Choose one to open a Message-Set Browser to access all the methods that implement the selected message.

**move** Moves the selected message to another protocol.

**remove** Removes the selected message from its class.

subview B

**again** as in text editing.

**undo** as in text editing.

**copy** as in text editing.

**cut** as in text editing.

**paste** as in text editing.

**do it** as in evaluating expressions.

**print it** as in evaluating expressions.

**accept** The text is a method that is compiled and stored in the method dictionary of the selected class under the selected protocol.

**cancel** Redisplays the text that appeared before any editing that occurred after the last **accept**.

**format** Redisplays the text, pretty-printed. Does not do an automatic **accept** since you might not like the formatted version. Does not work if the text has been edited since the last **accept**.

**spawn** Creates a Message Browser for the selected message in which the method is the edited (but not yet saved) version currently displayed. Does an automatic **cancel** in subview B.

**explain** Inserts an explanation of the single syntactic element that is the current text selection.

**accessible views** Message-Set Browser, Message Browser

### 5.2.5 Debugger

**name** Debugger

**general description** Used to explore the context of an error halt or breakpoint. Provides access to the sequence of message sends that terminated with access to this debugger.

The message-send sequence is displayed in a list menu in subview A in the form of

className>>message

or, if the message is found in a class other than className,

className (name of class whose method dictionary contained the message)>>message

The method associated with the message selected in subview A is displayed for editing in subview B. The instance variable names of the message receiver are shown in a list menu in subview C; the value of the selected variable is displayed in subview D (just as in an Inspector). The temporary variable names of the method shown in subview B are displayed in a list menu in subview E; the value of the selected variable is displayed in subview F (just as in an Inspector).

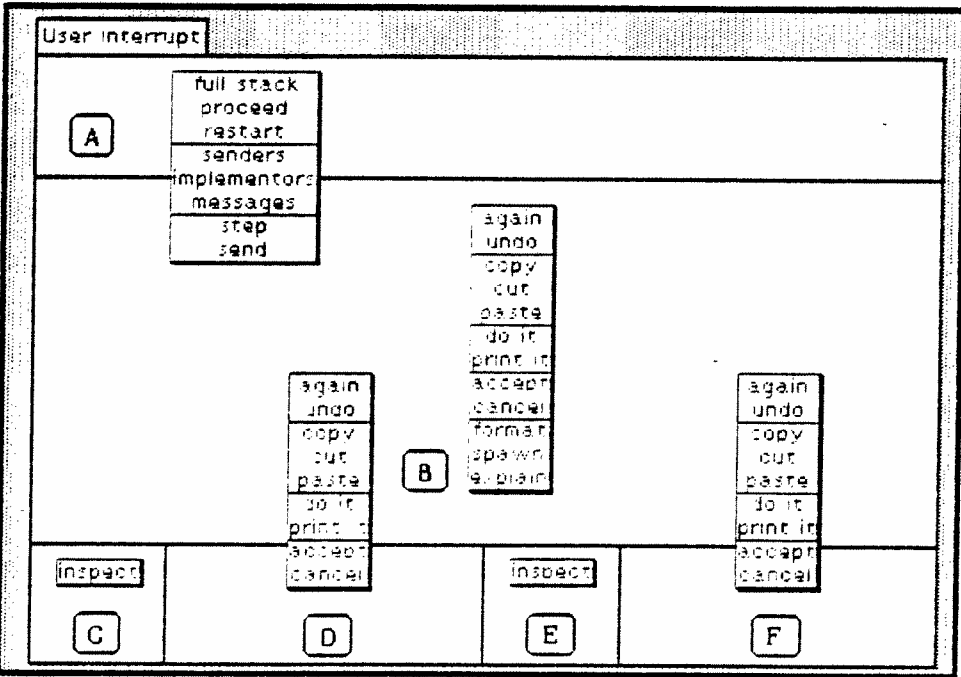
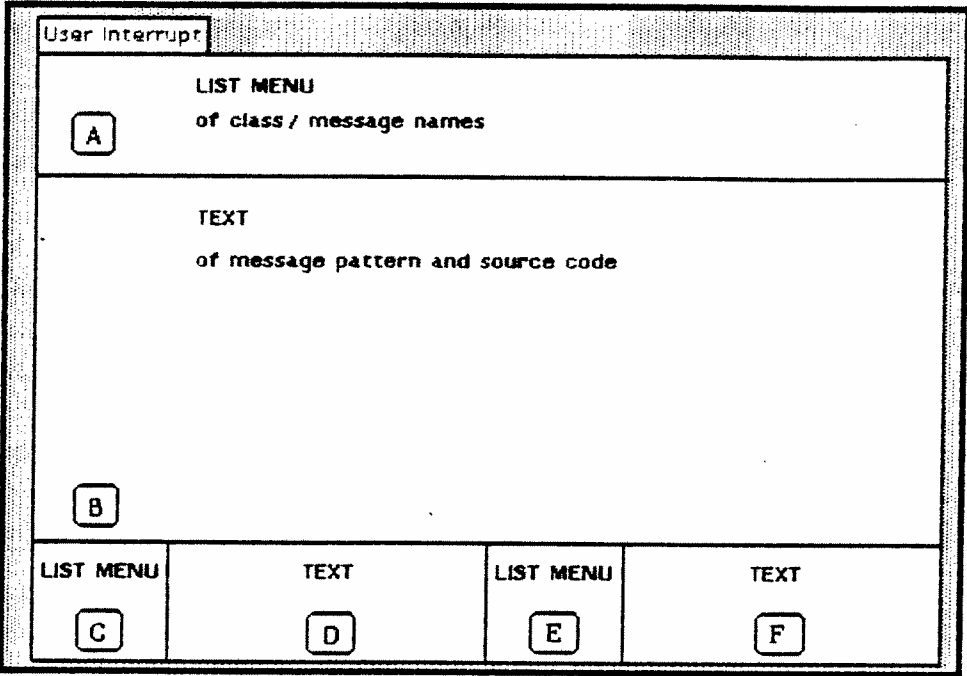
**how accessed** Notifier yellow button choose debug

**how created** Designate the rectangular area.

**how terminated** Blue button choose close.

**blue button activity** Default

**red button activity** Select items (methods) in the list menus in subviews A, C, and E; and select characters for text editing in subviews B, D, and F.



yellow button activity

subview A (no item selected)

fullStack

Opens the menu to the full stack of message-sends of the interrupted message-sending activity.

proceed

Closes the debugger and continues execution in the currently selected method. Proceed in the method of the last message-send in the sequence. The continuation assumes that the message at the point of the interruption had completed and determined a value. The value for proceeding is nil or the value of the last expression evaluated in subview B.

subview A (item selected)

senders

Opens a Message-Set Browser to access all the methods in which the selected message is sent.

implementors

Opens a Message-Set Browser to access all the methods that implement the selected message.

messages

Creates a menu of all the messages sent in the method associated with the selected message. Choose one to open a Message-Set Browser to access all the methods that implement it.

step

Continue evaluation to the next message-send in the currently selected method. If no method is specifically selected, assume the selection is the last message-send. Halts immediately after evaluating the next message-send.

send

If step were selected, evaluation would consist of a sequence of message-sends. Send evaluates only the next one of these and halts immediately after.

fullStack

Opens the menu to the full stack of message-sends of the interrupted message-sending activity.

restart

Closes the debugger and restarts execution from the beginning of the currently selected method.

proceed

Closes the debugger and continues execution in the currently selected method. The continuation assumes that the message at the point of the interruption had completed and determined a value. The value for proceeding is nil or the value of the last expression evaluated in subview B. Each time a new message-send is selected in subview A, the proceed value is reset to nil.

subview B

again

as in text editing.

undo

as in text editing.

copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions. Evaluation is carried out in the context of the interrupted activities at the state of the message-send selected in subview A. The value of the selected expression becomes the proceed value of the debugger.
print it	as in evaluating expressions.
accept	The text is a method that is compiled and stored in the method dictionary of the selected class under the selected category.
cancel	Redisplays the text that appeared before any editing that occurred after the last <b>accept</b> .
format	Redisplays the text, pretty-printed. Does not do an automatic <b>accept</b> since you might not like the formatted version.
spawn	Creates a Message Browser for the selected message in which the method is the edited (but not yet saved) version currently displayed. Does an automatic <b>cancel</b> in the subview.
explain	Inserts an explanation of syntactic elements in the current text selection.

subview C (no item selected)

No menu displays. Subview flashes.

subview C (item selected)

inspect

Opens an Inspector for the object referred to by the selected variable name. The variable **self** refers to the receiver of the message currently selected in subview A.

subview E (no item selected)

No menu displays. Subview flashes.

subview E (item selected)

inspect

Opens an Inspector for the object referred to by the selected variable name.

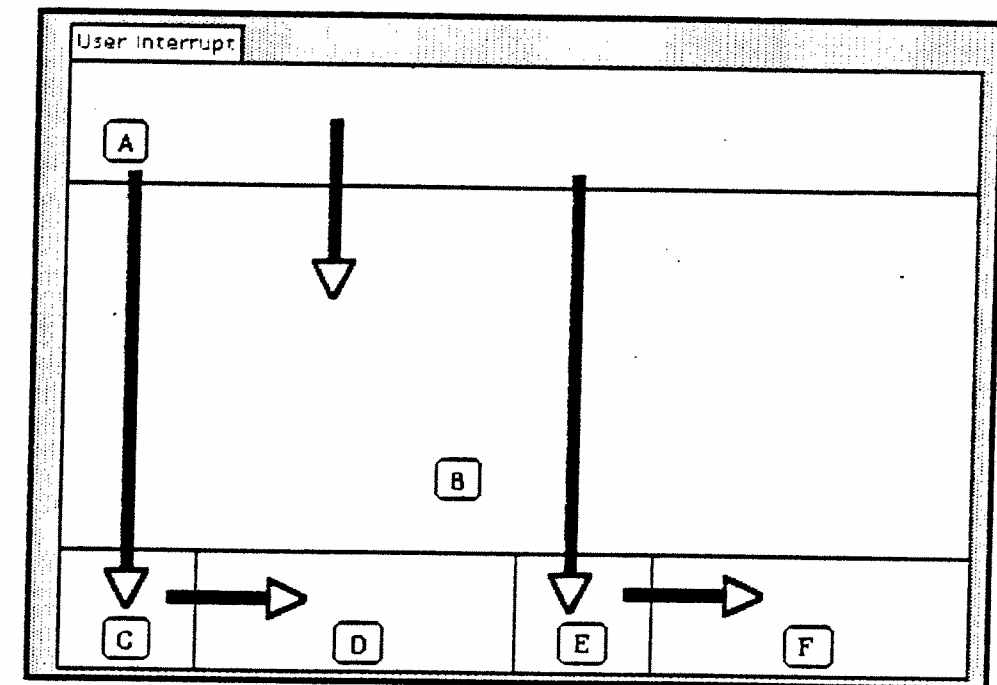
subview D and F

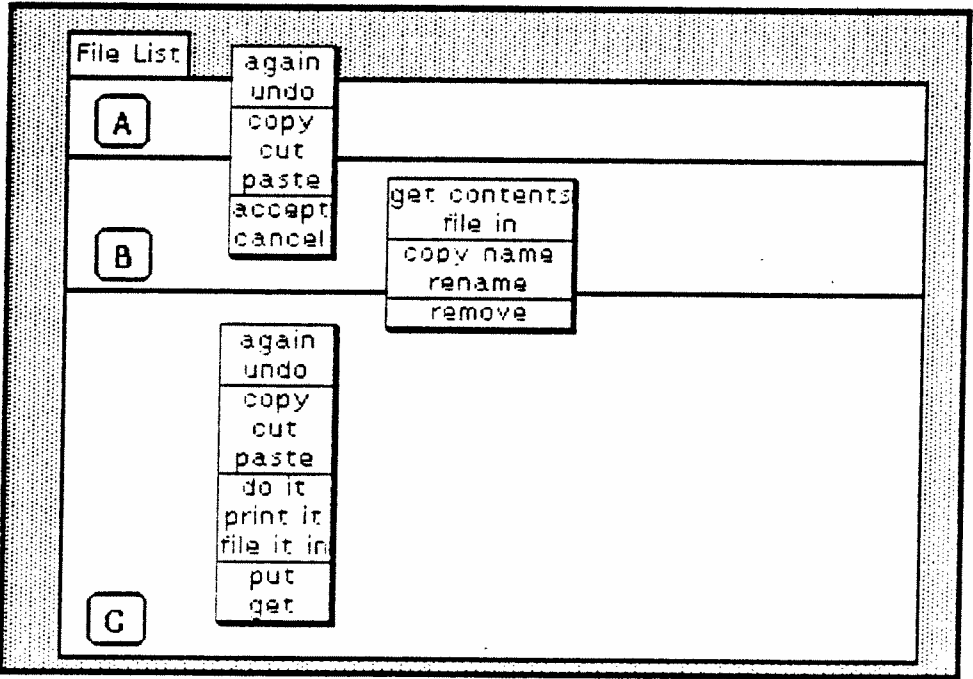
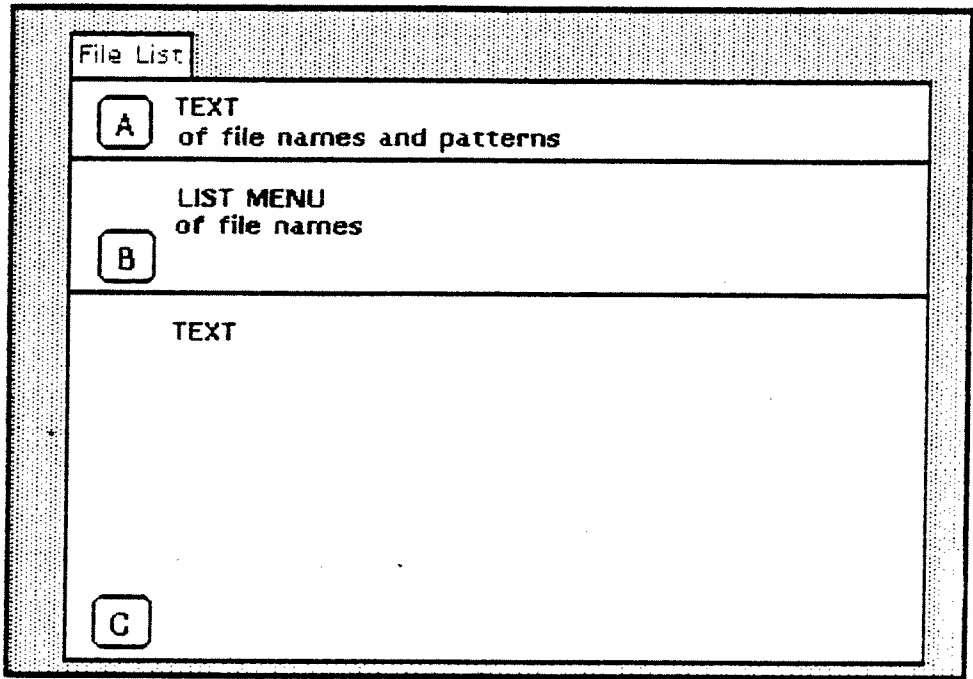
again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions. Evaluation is carried out

	in the context of the variables of the message-send selected in subview A.
print it	as in evaluating expressions. Evaluation is carried out in the context of the variables of the message-send selected in subview A.
accept	Saves the edited text as the actual text. In addition, saves the value of the text as the value of the currently selected variable, if any. Evaluation is carried out in the context of the variables of the message-send selected in subview A.
cancel	Redisplays the text that appeared before any editing was done after the last <b>accept</b> .

**subview dependencies** Text is displayed in subviews B, C, and E as the result of selection in A. The value in subview D is the value of the variable selected in subview C; the value in subview F is the value of the variable selected in subview E.

**accessible views** Inspector, Message Browser, Message-Set Browser

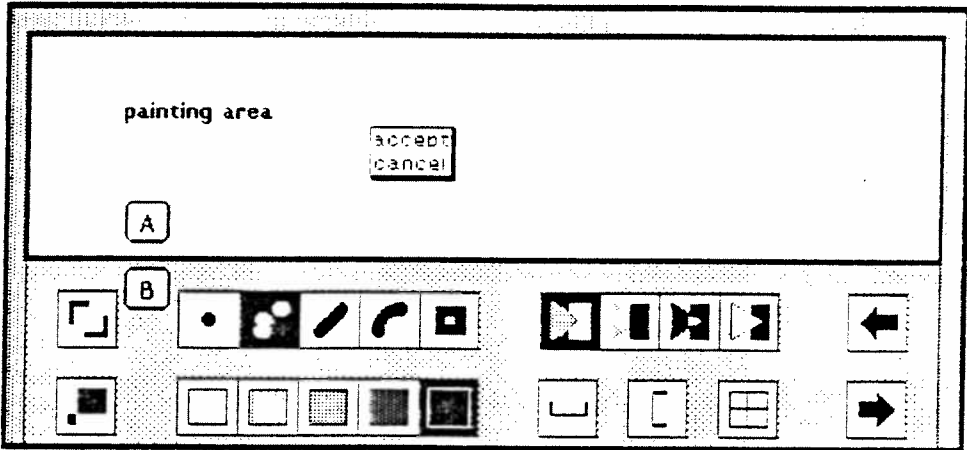
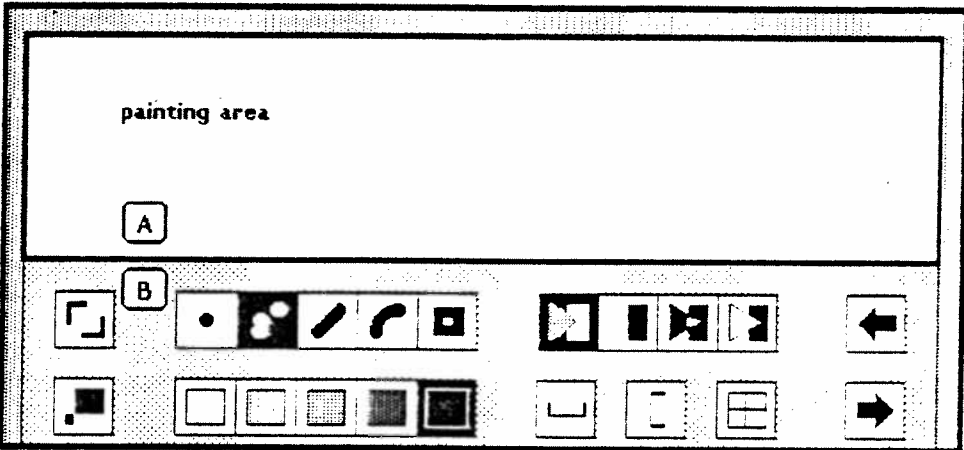
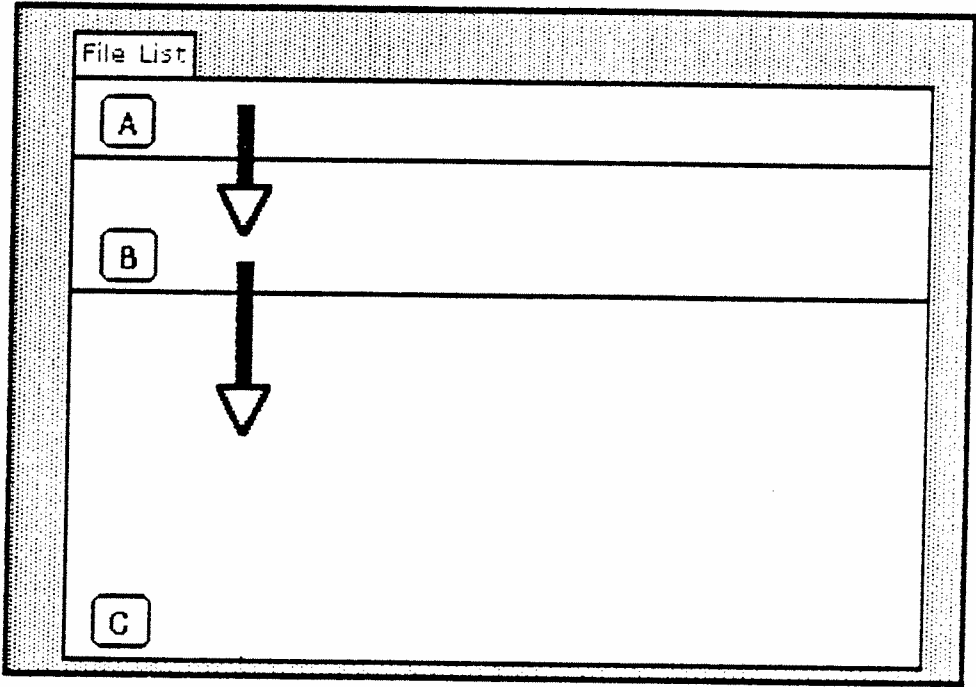




5.2.6 File List

name	File List
general description	Used to access contents of files on a file directory. User specifies which files should be referenced in a menu by naming files and/or naming patterns that match file names.
how accessed	1. System Menu choose file list. 2. Evaluate the expression FileList open
how created	Designate the rectangular area.
how terminated	Blue button choose close.
blue button activity	Default
red button activity	Choose items (file names) in the menu in subview B and select characters for text editing in subviews A and C.
yellow button activity	

subview A	
again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
accept	Saves the contents and then determines the alphabetic list of file names for subview B that match the contents; parsing assumes different names or patterns are separated by carriage returns.
cancel	Redisplays the text that appeared before any editing that occurred after the last accept.
subview B (no item selected)	No menu displays. Subview flashes.
subview B (item selected)	
get contents	Retrieve the contents of the selected file, and display the contents in the text subview.



file in	Retrieve the entire contents of the selected file, reading evaluating the text according to the file format for class descriptions and expressions.
copy name	Copy the text of the file name into the text editor buffer so that it can be "pasted" into other text views.
rename	Change the name of the selected file. A prompter appears, with the name of the selected file. Edit it with the new file name. Type the "carriage return" key or choose the yellow button command accept to indicate that you have completed the file name. The menu of file names will be updated. If you type an improper file name, or no file name at all, a confirmer will appear to determine whether you want to try again to specify a file name.
remove	Delete the selected file from the file directory. A confirmer appears to determine whether you really want to remove the selected file. Choose yes if you do, no if you do not.

subview C

again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.
print it	as in evaluating expressions.
file it in	Reads and evaluates the text selection.
put	Stores the text from the view onto the file.
get	Restores the text to the view from the file.

subview dependencies  
Text is displayed in subview C as a result of selection in subview B. The list displayed in B is the result of choosing accept in subview A.

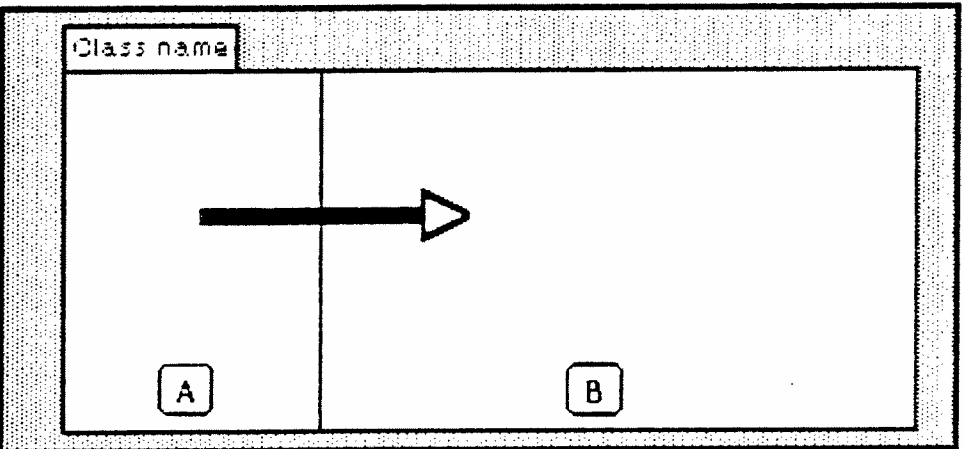
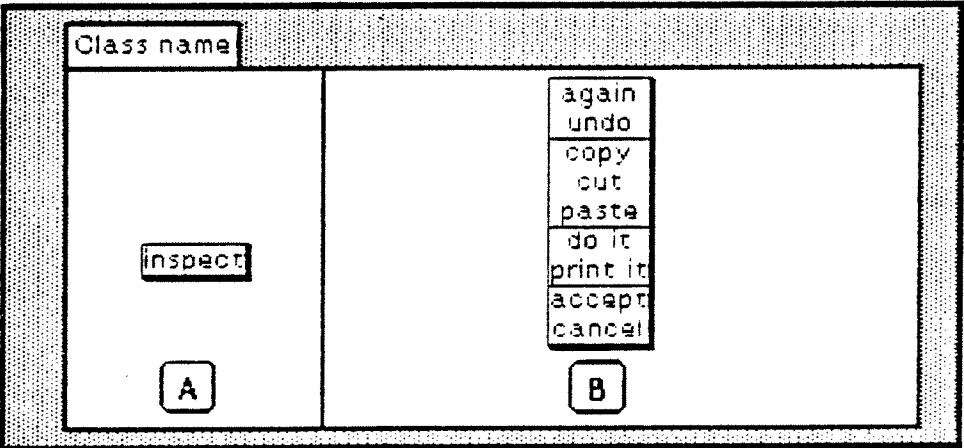
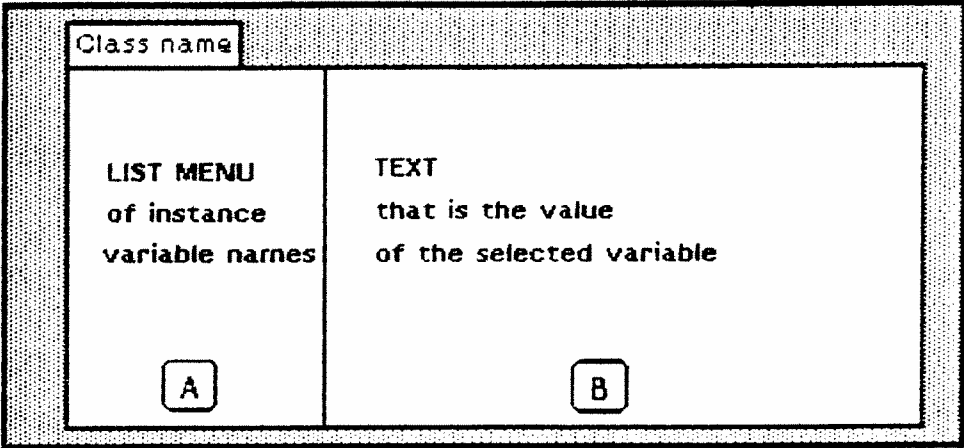
accessible views    Syntax Error View

5.2.7            Form Editor

The Form Editor is used to make pictures.

name              Form Editor

general description    Used to create and modify Forms.



how accessed

1. Evaluate one of the following forms of expression

Form fromUser edit

(Form new extent: extentPoint) edit

(Form new extent: extentPoint) editAt: originPoint

2. To be able to use the whole screen for editing,

FormEditor openFullScreenForm

how created

Designate the top left corner of the rectangular area. The extent of the area is fixed by the Form and Form menu sizes. In the case of the first expression in 1, you designate the Form's rectangular area first.

how terminated

Blue button choose close.

blue button activity

Default, although the frame size cannot be changed.

red button activity

Brush forms onto the "canvas" of subview A according to the painting tools. Select menu items in subview B to select the painting tools, modes, and halftones.

yellow button activity

subview A

accept	Save the current image as the Form.
cancel	Restore the saved Form as the current image.

keyboard activity

Each menu item corresponds to a key on the keyboard. Pressing the key is identical to selecting the menu item with the red button.

accessible views

Bit Editor (unscheduled)

5.2.8 Inspector

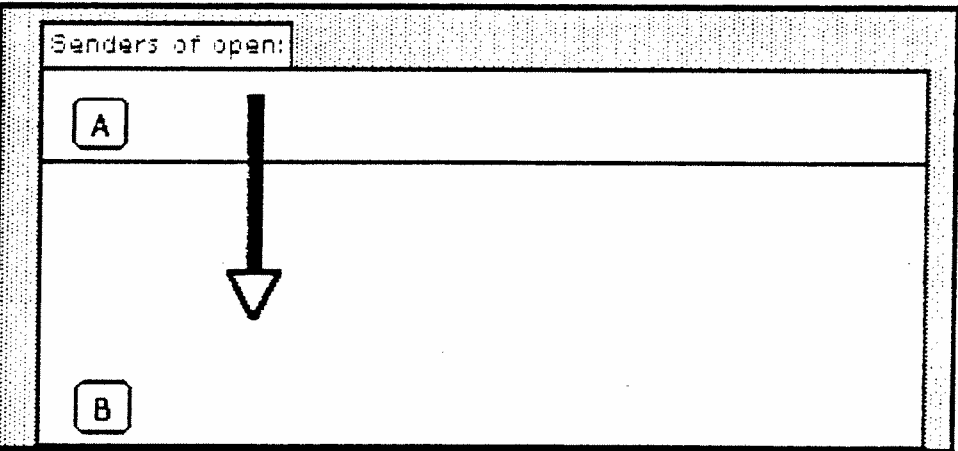
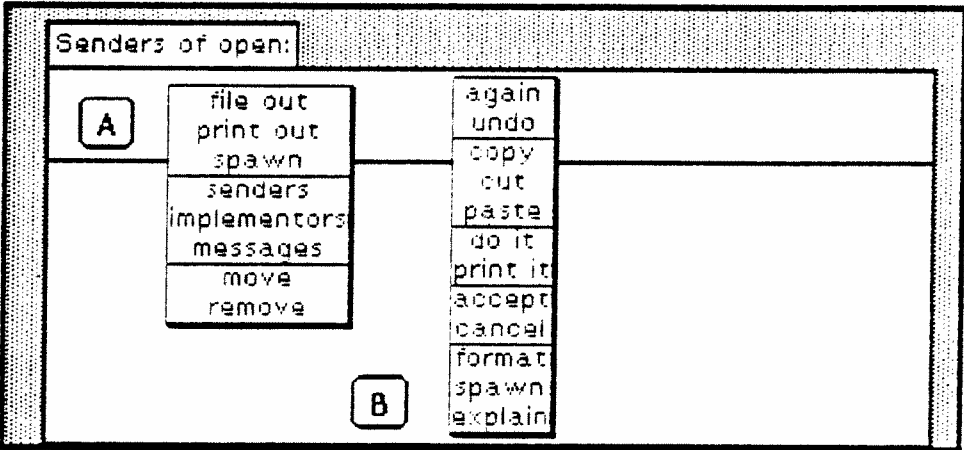
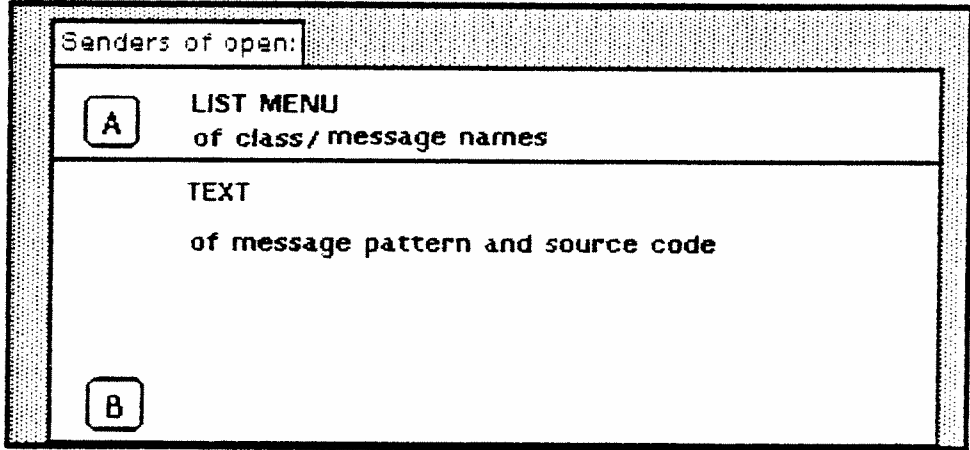
name

Inspector

general description

Used to examine the variables of an object.





- how accessed
1. Debugger subviews C and E choose inspect
  2. Inspector subview A choose inspect.
  3. Send any object the message inspect, for example, evaluate the expression  `#(2 4 6 8) inspect.`

how created      Designate the rectangular area.

how terminated      Blue button choose close.

blue button activity      Default

red button activity      Choose items in the list menu of subview A; select characters for text editing in subview B.

yellow button activity

subview A	inspect	Opens an Inspector for the object referred to by the selected variable name.
subview B		
	again	as in text editing.
	undo	as in text editing.
	copy	as in text editing.
	cut	as in text editing.
	paste	as in text editing.
	do it	as in evaluating expressions.
	print it	as in evaluating expressions.
	accept	Evaluate the text as a Smalltalk-80 expression. Replace the text by the resulting value. Store the value as the value of the currently selected variable, if any. If the text can not be successfully evaluated in the context of the variables of the object under inspection, an error is created. If no variable is selected in subview A and this command is chosen, subview B flashes.
	cancel	Redisplay the text that appeared before any editing was done after the last accept.

subview dependencies      Text is displayed in subview B as the result of selections in A.

accessible views      Inspector

5.2.9      Message-Set Browser

Message-Set Browsers appear in several places in the system, primarily to give access to senders and

implementors of messages.

**name** Message-Set Browser

**general description** Used to access a set of messages. The set is derived according to some retrieval criterion, such as: all messages whose associated method includes an expression whose message selector is identical to a particular selector. The "particular selector" referred to in the query is shown as the label of the browser.

**how accessed** 1. System Browser, System Category Browser, Class Browser, Message Category Browser, and Message Browser subview D yellow button choose senders, implementors, or messages.

2. Message-Set Browser subview A yellow button choose senders, implementors, or messages.

3. Change-Set Browser subview A yellow button choose senders, implementors, or messages.

4. Debugger subview A yellow button choose senders, implementors, or messages.

5. Specific collections can be browsed by evaluating the following forms of expressions. Expressions such as these appear in the System Workspace.

Smalltalk browseAllCallsOn: #keywordSymbol.

Smalltalk

browseAllCallsOn: #firstKeywordSymbol  
and: #secondKeywordSymbol.

Smalltalk browseAllAccessesTo: variableName.

<Class name> browseAllAccessesTo: variableName.

Smalltalk

browseAllCallsOn:  
(Smalltalk associationAt: #aSymbol).

Smalltalk browseAllImplementorsOf: #messageSelector.

Smalltalk browseAllSelect: aBlock.

**how created** Designate the rectangular area.

**how terminated** Blue button choose close.

**blue button activity** Default

**red button activity** Choose items in the list menus in subview A, and select characters for text editing in subview B. New messages defined in subview B will be categorized in the class and category of the current selection in subview A. If there is no such selection, then the code is simply ignored.

**yellow button activity**

Subview A and B in a Message-Set Browser correspond to subview D and E in a System Browser.

subview A (no item selected)

No menu displays. Subview flashes.

subview A (item selected)

file out

Prints the description of the selected message onto a file whose name is the class name followed by a hyphen followed by the selector and concatenated with the extension '.st'.

print out

Prints a "pretty printed" description of the selected message onto a file whose name is the class name followed by a hyphen followed by the selector and concatenated with a system-specific extension.

spawn

Opens a Message Browser in which only the description of the selected message for the selected class can be accessed.

senders

Opens a Message-Set Browser to access all the methods in which the selected message is sent.

implementors

Opens a Message-Set Browser to access all the methods that implement the selected message.

messages

Creates a list menu of all the messages sent in the method associated with the selected message. Choose one to open a Message-Set Browser to access all the methods that implement it.

move

Moves the selected message to another protocol.

remove

Removes the selected message from the message set and from its class. Confirmation will be requested.

subview B

again

as in text editing.

undo

as in text editing.

copy

as in text editing.

cut

as in text editing.

paste

as in text editing.

do it

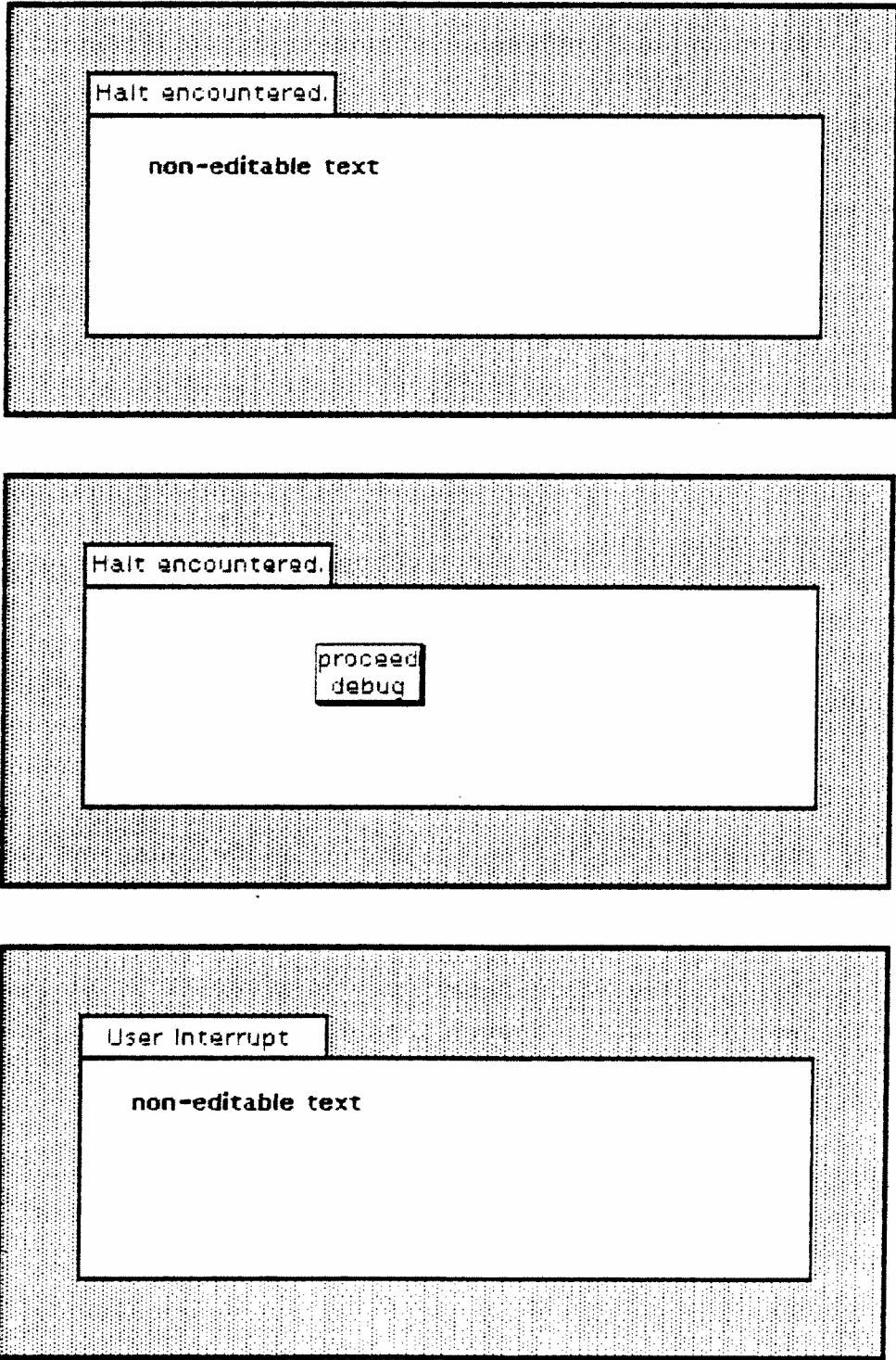
as in evaluating expressions.

print it

as in evaluating expressions.

accept

Typically, the text is a revision of an existing method that is compiled and stored back. Or the text is a new message and its associated method which will be stored in the class and category of the message selector selected in subview A.



cancel	Redisplays the text that appeared before any editing after the last accept.
format	Redisplays the text, pretty-printed. Does not do an automatic accept.
spawn	Creates a Message Browser for the selected message in which the method is the edited (but not yet saved) version currently displayed. Does an automatic cancel in subview B of this Message-Set Browser as well as the new Message Browser.
explain	Inserts an explanation of syntactic elements in the current text selection.

subview dependencies  
Text is displayed in subview B as the result of a selection in A.

accessible views  
Message Browser, other Message-Set Browsers

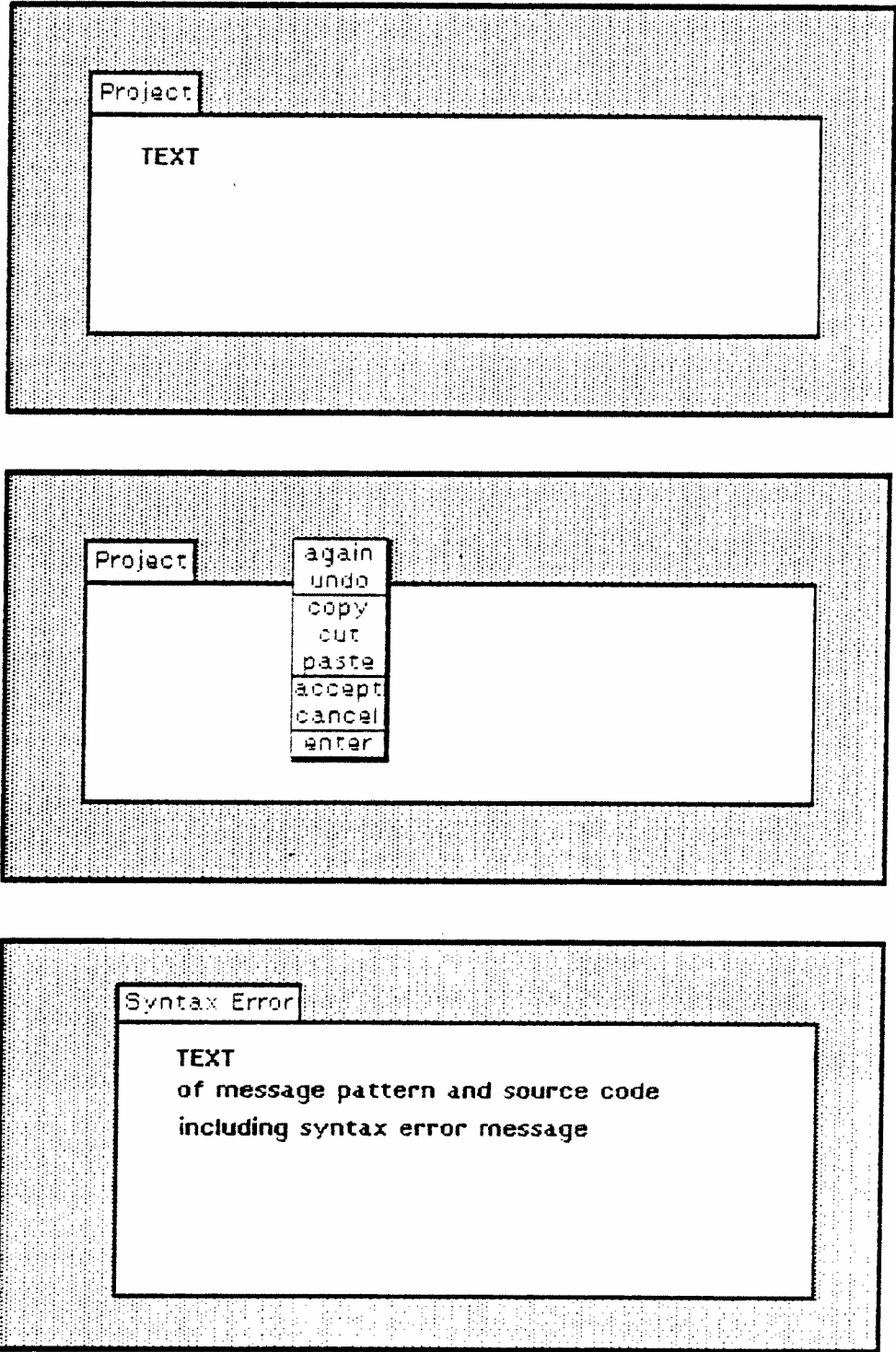
5.2.10 Notifier

Notification that a runtime error or user-inserted breakpoint has occurred appears in a notifier.

name  
Notifier

general description  
Used to inform the user that an interruption in activities has occurred. Text displayed indicates the last several message-sends before the interruption. You decide whether to continue or discontinue the activities. The decision can be deferred until you inspect the current state of the activities by opening a Debugger.

how accessed  
1. Evaluation interruption occurs due to an error that is typically the result of sending a message to an object that does not understand the message. The label of the view indicates which message was not understood; the first line of the text indicates the class of the unsuccessful receiver.  
  
2. Anticipated interruption occurs because the code includes an expression of the form  
  
self error: 'errorMessageString'  
  
The label of the view is the errorMessageString.  
  
3. Anticipated interruption occurs because, as part of debugging activities, you insert a breakpoint-code of the form  
  
self halt  
  
or  
  
self halt: haltMessageString  
  
The label of the view is either Halt encountered. or the haltMessageString.



4. You interrupt activities by pressing the "control" key and the "c" key on the keyboard at the same time. The label of the view is User Interrupt.

- how created
- Size and placement determined by the method that uses it. Typically appears centered in the currently active view.
- how terminated
- To ignore message but not continue the interrupted activity, blue button choose close.  
  
To ignore message and to continue the interrupted activity, yellow button choose proceed.  
  
To investigate the message using a Debugger, yellow button choose debug.

blue button activity Default

red button activity None

yellow button activity

- proceed
- The action interrupted by the appearance of this message is continued. Note, this might be inappropriate if an error occurred because an object is in an improper state.
- debug
- Opens a Debugger on the context of the interrupted action.
- correct
- When a Notifier occurs as a result of a runtime error that a message was not understood, this command is included in the menu so that you can ask the system to try to guess an appropriate message-selector substitution.

accessible views Debugger

5.2.11 Project

name Project

general description An individual screen can contain several views of information: for example, browsers, workspaces, transcripts, inspectors, debuggers, and projects. The system can have several individual screens. These are called *projects*. Each project maintains its own set of scheduled views and its own record of changes that you make to the class definitions.

The project view serves as an entry to a project. It appears as a rectangular area in which text can be prepared, just as in a workspace. This text describes the project for documentation purposes. "Entering" the project puts your work in the context of this project's set of scheduled views and changes.

how accessed	System Menu choose project.
	Projects form a hierarchy for accessing purposes. Each project can contain views through which sub-projects can be accessed. One project was created when the system was initialized. It is maintained as the "top" of the project hierarchy.
how created	Designate the rectangular area.
how terminated	Remove a project using blue button choose close. This will produce a confirmer if any views or changes still exist in the project, or if the project description has not been saved.  To leave a project, choose System Menu command exit project.
blue button activity	Default
red button activity	Select characters for text editing.
yellow button activity	
	again as in text editing.
	undo as in text editing.
	copy as in text editing.
	cut as in text editing.
	paste as in text editing.
	accept Stores the edited text as the actual text.
	cancel Redisplays the text that appeared before any editing was done after the last accept.
	enter Leaves the current project and enters the context of the project referred to by this view. Refreshes the screen with the new project's views.

### 5.2.12 Syntax Error

Syntax errors in files result in a special error notification called a Syntax Error View.

name	Syntax Error
general description	Used to present a syntax error encountered when reading a method or expression from a file. The error can be corrected, the method compiled and saved or the expression evaluated, and then reading the file resumed.
how accessed	1. File List view yellow button choose file in. 2. Evaluate an expression of the form

(FileStream file: 'fileName') fileIn.

when, in each case, an error occurs.

how created Designate the rectangular area.

how terminated Blue button choose close.

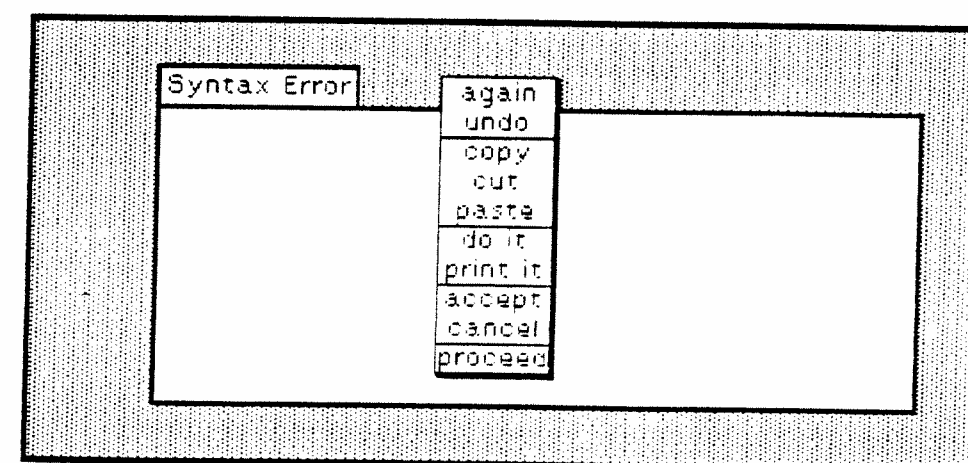
blue button activity Default

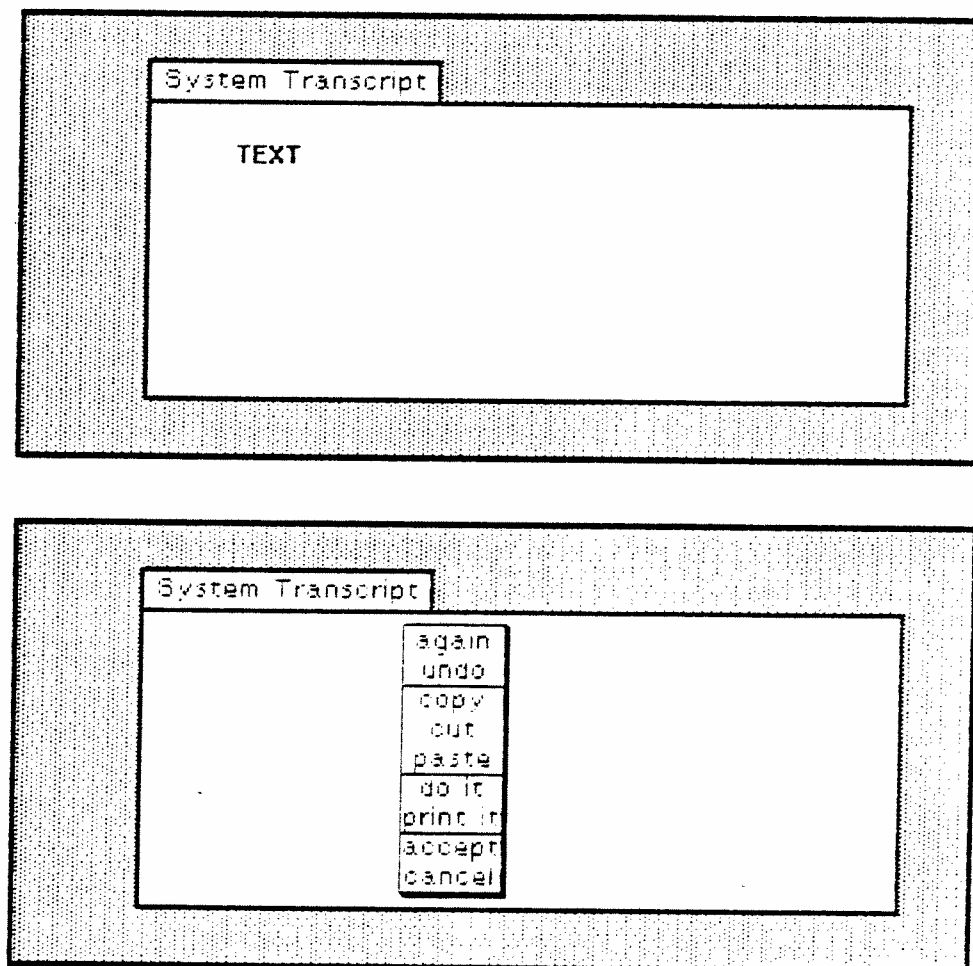
red button activity Select characters for text editing in subview A.

yellow button activity

subview B

again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.
print it	as in evaluating expressions.
accept	The text is a method; compile it and store it in the appropriate class. Automatically proceed to read the file.
cancel	Redisplays the text that appeared before any editing after the last accept.
proceed	Continues to read the selected contents of the file.





### 5.2.13 Text Collector

The primary Text Collector in the system is the System Transcript.

**name** Text Collector (Transcript)

**general description** Like a workspace, except it is also possible to send messages that store characters in the view's text. The message protocol is similar to that for storing characters in any WriteStream.

**how accessed** 1. System Menu choose system transcript.

One special text collector is maintained and labeled the "System Transcript." It is referred to by the global variable Transcript. Choosing system transcript creates a(nother) view of this particular text collector.

2. In order to create and schedule a new text collector, evaluate the expression

```
TextCollectorView
  open: TextCollector new
  label: 'TextCollector'
```

**how created** Designate the rectangular area.

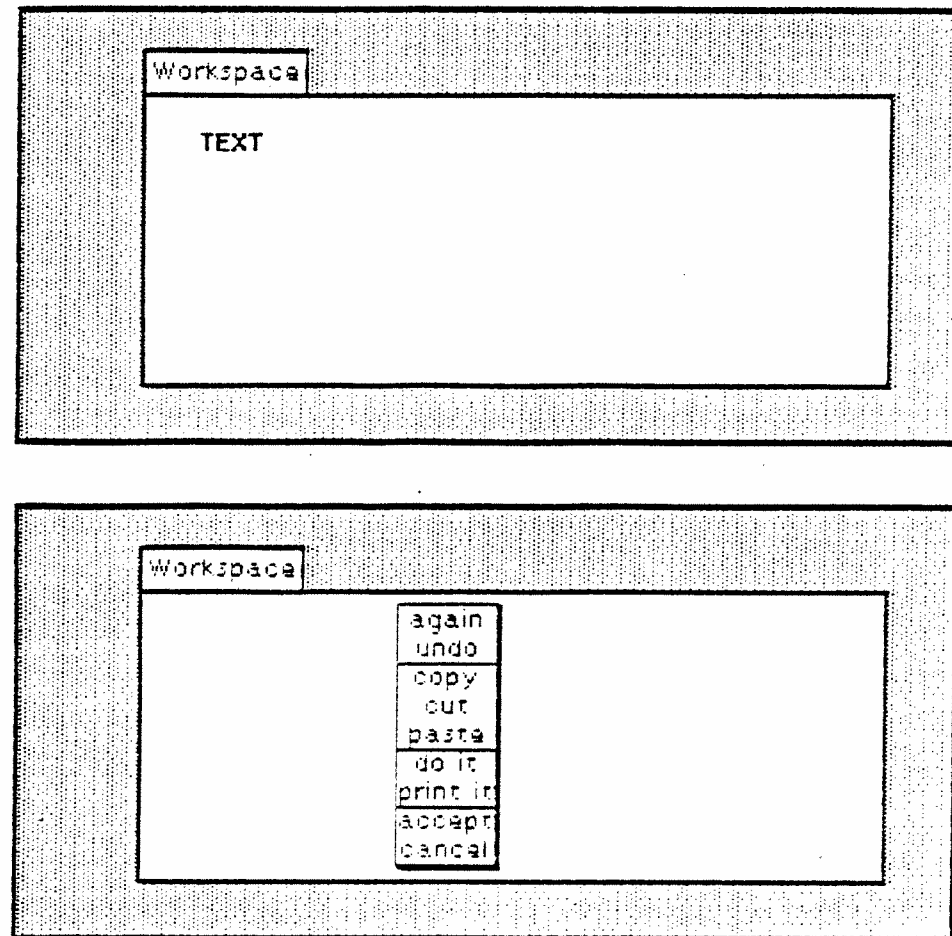
**how terminated** Blue button choose close.

**blue button activity** Default

**red button activity** Select characters for text editing. Note that changes due to text editing are not reflected in multiple views of the System Transcript, although changes due to message-sending are.

**yellow button activity**

again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.
print it	as in evaluating expressions.
accept	Store the edited text as the actual text.
cancel	Redisplay the text that appeared before any editing was done after the last accept.



## message-sending

nextPut: aCharacter	Store the character.
nextPutAll: aString	Store each character in the string.
print: aString	Same as nextPutAll: aString.
show: aString	Store each character in the string and redisplay the view.
store: anObject	Store a description of the argument, anObject, in a form that, when executed, reconstructs the object.
space	Store the character representing a space.
cr	Store the character representing positioning to the beginning of the next line of text.
crtab	Store the character representing positioning to the beginning of the next line of text followed by a tab.
tab	Store the character representing a tab.
crtab: anInteger	Store the character representing positioning to the beginning of the next line of text followed by anInteger number of tabs.
clear	Remove all characters and display the empty view.

## 5.2.14 Workspace

Access to simple text editing is provided in a Workspace. To explore ways to access the text editor, examine system classes DisplayTextView, StringHolderView, and TextView, as well as class ParagraphEditor.

name	Workspace
general description	Used as "paper" on which to prepare text. An original version and edited (working) version are maintained. If you attempt to close the view when the text has not been stored (yellow button choose accept) then a confirmation is required.
how accessed	System Menu choose workspace.  One special workspace is maintained and labeled the "System Workspace." It is a class variable of class StringHolder. Copies of the system workspace can be opened by choosing System Menu system workspace.
how created	Designate the rectangular area.



how terminated    Blue button choose close.

blue button activity Default

red button activity Select characters for text editing.

yellow button activity

again	as in text editing.
undo	as in text editing.
copy	as in text editing.
cut	as in text editing.
paste	as in text editing.
do it	as in evaluating expressions.
print it	as in evaluating expressions.
accept	Saves the edited text as the actual text.
cancel	Redisplays the text that appeared before any editing was done after the last accept.