How To Use the Smalltalk-76 System

Learning Research Group
October 1979

(if you find errors, etc, inform Adele Goldberg)

For Internal Xerox Use Only

XEROX

Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Table of Contents

I. Getting Started, Text Editing, and Printing

This and subsequent documentation assumes you are familiar with the Alto, Ethernet, and mouse pointing device. If not, you should obtain separate documentation and read it before continuing with this material. This document does not purport to teach Smalltalk, the programming language; it only presents the user interface.

Getting Small

Your first step is to set up your own Smalltalk disk. The easiest way to do this is by copying the Basic Smalltalk Disk provided for Smalltalk courses. The course disk in Adele's office, or the Basic Smalltalk disk usually available in Dan Ingalls' office, contains all the files you will need to run Smalltalk, along with FTP (for transferring files from one place to another over the Ethernet), and Empress (for printing files in hardcopy form).

All important Smalltalk-related files are kept on the Ivy file server in a directory named <Smalltalk>. The latest version of the system is always available in this directory under the name *Small.boot*; there is also a corresponding symbol file called *Smalltalk.syms*, containing symbol-table information needed by the Swat debugging utility. These two files should always be retrieved together. An Alto command file for this purpose is kept in the [Ivy]<Smalltalk> directory as *GetSmall.cm*: if you keep a copy of this file on your disk, you can always obtain the latest Smalltalk release by typing

```
@GetSmall cr
```

to the Alto Executive. You should be running under OS-16. Smalltalk's prior to version 5.5 run only under OS 15.

Whenever a new version of Smalltalk is released, the previous version is saved on the files *OldSmall.boot* and *OldSmalltalk.syms* -- if you encounter difficulties with a new release, you can back up to the previous version by retrieving these files with the command file *GetOldSmall.cm*. If you have an Alto II with the XM, or Extended Memory feature, you can use the special XM version of Smalltalk, designed to make use of the extra memory available. This version, stored on *XMSmall.boot* and *XMSmalltalk.syms* and accessed via *GetXMSmall.cm*, gives noticeably better performance than the standard release, but is updated less frequently and often lags in incorporating new features and corrective changes.

Sources are accessed from the Woodstock server located in Palo Alto, California. A local copy can be obtained by using FTP to get [IVY]<SMALLTALK> Smalltalk.Sources, and then executing in Smalltalk

```
user sourcesTo: (dp0 file: 'Smalltalk.Sources') changesOnly: true.
```

The code to do both the FTP and the message to user can be found in the user workspace window as described later.

Entering and Leaving SMALLTALK

To enter Smalltalk from the Alto Executive, type

Resume Small.boot cr

To leave Smalltalk and return to the Executive, evaluate the expression `user quit`

either by typing it into the dialog window, as described below under THE DIALOG WINDOW; by selecting and executing it in a code window or code pane (see the section: CODE WINDOWS, Execution and Compilation); or by selecting the command `quit` in the project view menu (see section: PROJECT VIEW WINDOWS in Section 2). A line containing this message is included for your convenience in the "UserView workspace" window provided in the Smalltalk release (initially, the upper right window on the screen). Whenever you quit Smalltalk in this way, the current state of your working environment is saved on the file *Small.boot*; the next time you resume *Small.boot*, you will find the Smalltalk system in almost the state in which you left it (files and ether are reset). Thus there is no need to save your work at the end of one session or to restore your working context at the beginning of the next: you simply quit and resume, with no break in continuity. (Of course, if you retrieve a fresh copy of the Smalltalk release from Ivy, your old working environment will be lost. If you have made changes to the system that you wish to save, you must explicitly file them out before obtaining the fresh release and file them back into the new copy--see Section 4.)

Windows and Menus

All communication with Smalltalk is done through a *window*. Windows come in various flavors, each with its own properties and capabilities, and can be freely created, destroyed, and moved around on the screen. Only one window is "awake" at any given time. To "wake up" a window, place the cursor into it and press any of the mouse buttons: the window will redisplay its image on the screen to signify that it is awake, possibly overlaying and completely or partially hiding some of the other windows; the window that was previously active will "go to sleep." (If any of the "flashers" at the top of the screen are active, Smalltalk is busy doing something else, such as disk management, and may not hear your wakeup signal immediately--persevere.)

When you obtain a fresh copy of Smalltalk from Ivy, you will find three windows initially displayed on the screen. Prominent in the center is a *browse window* (which is a Smalltalk object of class `BrowseWindow`); in the upper left corner is a *dialog window* (class `Dispframe`); at the upper right, overlapping and partially hiding the browse window, is a *code window* (class `CodeWindow`) titled "UserView workspace". When you have learned how to use the browse window, you can use it to inspect the code that drives any of these windows (including the browse window itself) by browsing the appropriate class. (See Section 2).

THE DIALOG WINDOW

Those who have previous experience with conventional interactive language systems will find the behavior of the dialog window the most familiar. (Note that its text is not editable and therefore its use is discouraged in favor of editing and executing in a code window.) Enter the window with the mouse and press any of the mouse buttons. The window will refresh its image and prompt you with the character to let you know it is listening. You can now proceed to hold an interactive dialog with Smalltalk. Keeping the cursor in the dialog window, type any Smalltalk expression, terminated by a *line-feed character*, which will appear as the Smalltalk graphic ("do-it"). Smalltalk will respond with the value of the expression you typed, and will prompt you with another . Try typing `3 + 4` into the dialog window--you will not be disappointed. If you type an expression ending with a period, it will be evaluated strictly for effect: the value it returns will be discarded and the default value `nil` displayed instead. This is useful for the cases in which either no result is desired (such as clearing the screen) or the result has a very extensive response to the message `print` which is not desired.

The dialog window offers limited editing facilities for correcting typing errors: backspace cancels the last character, control-w cancels the last word. If the window becomes full, it will scroll up automatically to make room for another line of dialog; there is no way to scroll down in order to inspect something you typed earlier. To clear the dialog window, type control-x: the window will become empty and prompt with a new . If the cursor leaves the dialog window, the window will continue listening to the keyboard (provided you don't put it to sleep by waking up another window), but will stop echoing the text you type in and will not respond to the "do-it" character . Reentering the dialog window will cause it to display all the text you have typed since leaving the window, and to respond to any do-it's this text may contain.

KEYBOARD/CHARACTER CORRESPONDENCES

Character	Smalltalk KeyStroke	Meaning
	ctrl a or ctrl ,	relational
	ctrl n or ctrl \	relational
	ctrl r or ctrl .	relational
	ctrl f or ctrl =	relational
	ctrl :	remote evaluation
	ctrl /	then only
	ctrl g or ctrl [	subscript
	ctrl '	literal
	ctrl _ or ctrl q	return
	ctrl]	point
	ctrl u or shift -	high minus

CODE WINDOWS

Because they provide more flexible editing facilities, code windows are generally more convenient than the dialog window for communicating with Smalltalk. As you become more familiar with the system, you will probably find yourself using them more and more and the dialog window less and less.

Scrolling

Move the mouse into the code window at the upper right titled "UserView workspace", and press one of the mouse buttons. You will see the workspace window wake up, and the prompt character will disappear from the dialog window, signifying that that window has gone to sleep. A *scroll bar* will appear at the left of the workspace window to show that the window is awake and ready to respond. If you move the mouse out of the window, the scroll bar will go away. This means that the window is not prepared to respond to your requests, although it will remain awake until you wake up some other window -- when you reenter the window the scroll bar will reappear. Move the mouse into the scroll bar: the cursor will change into an upward-pointing arrow when you are in the right half of the scroll bar, a down arrow in the left half. When you leave the scroll bar, the cursor will resume its normal form, a "northwest arrow."

Enter the scroll bar with the mouse and press the "red" (left or top) button. The line of text directly opposite the cursor will jump to the top of the window, if you are in the right half of the scroll bar (indicated by an up arrow cursor); the first line will move down to the cursor position, if you are in the left half of the scroll bar (indicated by a down arrow cursor). The window will continue to scroll repeatedly in this way as long as you hold down the button--you can use the vertical position of the cursor within the scroll bar to control the speed at which the window scrolls.

The small box within the scroll bar is a *position indicator*, whose vertical location shows the location of the window's visible contents relative to its text as a whole: the top of the scroll bar represents the beginning of the text, the bottom represents the end. If you move the cursor into the position indicator itself, the cursor will change to a small dot. By holding down the red button and moving the mouse, you can then drag the position indicator to any desired location within the scroll bar. When you release the button, the window will scroll to the corresponding position in the text.

Selection (red mouse button)

Most operations in a code window are based on the idea of *selecting* a position or a passage in the window's displayed text and manipulating it in some way. The system identifies the passage currently selected by highlighting it on the screen; if the selection is a position between characters, it appears as a vertical bar at the appropriate location. Selections are made with the red mouse button (left or top), according to the following conventions:

- To select a position between characters, simply point at the desired position and click the button.

- To select a whole passage, point at the beginning of the passage, press and hold the button, and move the cursor to the end of the passage before releasing the button.

- To select a single word, point anywhere within the word and click the button twice. (Timing here is not relevant as no clock or timer is involved; what matters is the sequencing of two clicks with the cursor pointing to the same location. Just give the system enough time to notice the first click before executing the second one.) For purposes of this rule, a word is any sequence of "tokenish" characters bounded by nontokenish characters. (Tokenish characters are letters, digits, period, colon, and the special Smalltalk characters "open colon" and "high minus" .) If you double click in this manner at the space between words, the word selected is the one closest to the vertical bar.

- To select a whole line, point at the beginning or end of the line and click twice. A line is delimited by a carriage return which has no visible printing character. It may seem that several "lines" are selected due to the automatic text wrap-around in a window.

- To select everything between a pair of matching left and right delimiters, point just inside either delimiter and click twice. Delimiter pairs on which this will work are parentheses ( ), square brackets [ ], angle brackets < >, curly braces { }, single (string) quotes ’ ’, and double (comment) quotes " ".

  Left delimiter takes precedence over the right one. That is, should you select between two delimiters, the choice is the one on the left. For example,

  (’ABCD’) EFGH’IJ’

  If you select between the ’ and), the matching assumes you mean to pair single quote marks; the selection is)EFGH .

- To select the last passage just typed from the keyboard, press the escape key.

Editing (yellow mouse bug menu and typing)

When the window is awake and contains the cursor, any text typed on the Alto keyboard will automatically replace the current selection; if the selection is a position between characters, the effect is to insert the typed text at that point. Pressing the delete key will delete the passage currently selected; backspace deletes the character immediately preceding the current selection; control-w deletes the word preceding the selection. (These last three conventions can also be used to correct errors during type-in.) Further editing facilities are available through the code window’s yellow-bug (middle mouse button) menu (i.e., press yellow bug, hold the button until the item is selected and then release the button):

- The menu command `cut` deletes the current selection, and is equivalent to the delete key on the keyboard.

- After a passage has been deleted or cut, it can be moved to another location by selecting

the new location and issuing the command `paste`.

- A piece of text can be copied to a new place without deleting it from its original location by selecting the text to be copied, invoking the menu command `copy`, then selecting the destination location and invoking `paste`.

- In general, the `paste` command replaces the current selection with a copy of the last piece of text removed using the delete, cut or copy command, or text typed in from the keyboard. Thus, to insert the same text in several places, select the first location and type the desired text, then select each of the other locations in turn and invoke paste in each place.

- After a passage is selected and replaced, the command `again` will find the next occurrence of the same passage and repeat the replacement.

- The command `undo` rescinds the last delete or paste operation and returns the text to its previous state.

These operations are not limited to a single window: for example, you can cut or copy a passage from one window and paste it down in another.

Formatting (yellow mouse bug menu and control keys)

Some limited formatting capabilities are provided in the code window. The yellow-bug menu includes the command `align`. Success selection of this command justifies the left margin, right margin, both margins, or not both margins (centered text).

Font changes are made using control characters. Make a selection in the window, then type

ctrl 0 - ctrl 9	to select the font specified in *DefaultTextStyle*
ctrl i	to select italics
ctrl b	to select bold
ctrl -	to select underlining
ctrl x	resets to font 0, no bold, no italic

Italics, bold, and underlining are removed by typing: shift ctrl i, shift ctrl b, or shift ctrl `-`, respectively. These changes are not made during type-in.

Execution and Compilation (yellow mouse bug menu)

A code window can be used to do anything you can do in the dialog window, with the added advantage that you can edit your input with the facilities just described. To evaluate a piece of text displayed in a code window as a Smalltalk expression, select the desired text using the selection conventions given above, then issue the menu command `doit`. Smalltalk will black out most of its screen while evaluating the expression. When the screen returns, the value of the expression will be inserted after the expression.

Try this at your own Alto: with the UserView workspace window awake, select a convenient location in the window and type `3 + 4.` Then hit the escape key to select the text you have just typed: it will be highlighted on the screen for identification. Now press the yellow mouse button to display the menu, do not release the button until you have selected the command `doit`, and then release the button. The expected result will appear adjacent to the text previously selected; it will be selected so you can delete it immediately if you choose. The result can also be placed in other positions in the text using the `paste` command.

The command `compile` in the code window's yellow-bug menu is used to finalize the window's edited text and make it permanent. Every code window is associated with a Smalltalk message; the window contains the *method* (program) for that message. The `compile` command recompiles the method, incorporating the changes you have made as a permanent part of your Smalltalk system. The menu command `cancel` wipes out all changes made since the last compile, restoring the window's text to its previous state.

The UserView workspace is a view of a message `workspace` in class `UserView`. Executing `compile` will change that method in your Smalltalk system (but not in the sources stored on the network).

Editing a Disk File

In order to create a code window displaying the contents of a new or old disk file, evaluate an expression of the form

```
(dp0 file: 'filename') edit.
```

or

```
f edit
```

where `f` is an object of class `FileStream,` either by typing in a dialog window, or typing and selecting in a code window as described above. (Some caveats: since the file is stored internally as a single *String*, editing can be rather slow and the size is limited to 16384 characters; when you store (see below), the text is saved, but not the formatting.

When you issue this command, the last active window will go to sleep and the cursor will assume the form of an *origin cursor* -- a right angle representing the new window's top left corner. Move the cursor to the desired position for this corner of the window, then press and hold any of the three mouse buttons; the cursor will change to a *corner cursor* representing the bottom right corner of the window. As long as you continue to hold the button, the new window will display a flashing frame, with its top left corner at the location you have designated and its lower right corner following the movements of the cursor. When you release the button the window's location will be frozen, its contents will be displayed, and the cursor, resuming its normal form, will automatically be repositioned to the center of the window.

The window is a code window and therefore all editing is carried out as before. However, there

are two changes to the yellow bug menu. The `compile` and `cancel` commands are replaced by `get` and `put`. To save your edits (but not your formats), select `put`; to retrieve the file contents prior to your edits since the last `put`, select `get`.

Printing the Disk File

Printing is described in detail in Section 4. For now, in order to complete your assignment, you can print the file by selecting the command `print` in the menu that appears when you hold down the mouse's blue button (right or bottom button). When the execution of the command is completed, the file should have been sent to the printer. If it hasn't, you can get out of Smalltalk (by executing `user quit`) and print the file called *window.press* using *Empress.*

EDITING WINDOW TITLES

The titles to most windows are editable using selection and typing only. Invoking yellowbug menu results in a simple flash of the title frame. Windows on to files do not have editable titles.

Document Editor

It is possible to prepare and print a document that integrates textual and pictorial images using the Smalltalk document editor. Good printing and formatting capabilities are provided. The explanation for using this editor is omitted from this documentation, however, because of the changing nature of this experimental system. Look at classes organized under the category `FPI Packages.`

2. The Smalltalk-76 User Interface: Text, Picture, Browsing and Project Windows

Windows and Menus

Most windows offer two different menus. The yellow (middle) mouse button activates a menu of commands specific to the particular kind of window, representing operations the window knows how to perform on its own contents as displayed on the screen; the "blue" button (the one on the right or the bottom, depending on the kind of mouse you have) invokes a menu of commands shared by all windows, which are used for manipulating the windows themselves on the Alto screen. These "blue-bug" commands are discussed below**.** The yellow-bug commands must be described separately with respect to each kind of window. We have already done so for code windows and windows onto file streams (see section 1).

BLUE-BUG COMMANDS

For the most part, manipulation and positioning of windows on the Smalltalk display is done through the blue-bug menu, which is the same for all kinds of windows. (Some of the more exotic kinds of windows have their own blue-bug menus, which we will not discuss here; the dialog window is rather special, and has no menus at all.) On the standard blue-bug menu are the following commands:

```
under
frame
close
print
printbits
```

- `under` wakes up and brings to the top the "bottommost" window under the current position of the cursor. This command is useful for getting your hands on a window that has become "lost" under another window, which hides it completely. (You don’t need the `under` command to wake up a window that is only partially obscured: simply enter the visible portion of the window with the mouse and press one of the buttons.)

- `frame` is used for changing the size and location of a window. You must specify the window’s new location in the following way. When you issue this command, the cursor will assume the form of an *origin cursor* -- a right angle representing the new window’s top left corner. Move the cursor to the desired position for this corner of the window, then press and hold any of the three mouse buttons; the cursor will change to a *corner cursor* representing the bottom right corner of the window. As long as you continue to hold the button, the new window will display a flashing frame, with its top left corner at the location you have designated and its lower right corner following the movements of the cursor. When you release the button the window’s location will be frozen, its contents

will be displayed, and the cursor, resuming its normal form, will automatically be repositioned to the center of the window.

- `close` destroys a window permanently and removes its image from the screen.

- `print` and `printbits` are both intended to generate press files containing a representation of the window's image as it appears on the screen. The difference between the two commands is that `printbits` represents the window's image in pure bit-map form, whereas `print` uses a combination of graphical and text formats. For more details on printing, see Section 4.

CREATING NEW WINDOWS

New windows can be created on the screen in a variety of ways, depending on the type of window desired. All of them require that you specify the location of the new window on the screen by the method described for the `frame` command in the window blue-bug menu.

You can create a `CodeWindow` for editing the code of any desired method by sending the message `edit:` to the class in which the method is defined. For example, to edit the method for message `twiddle` in class `Mumble`, evaluate

```
Mumble edit:  twiddle
```

Notice that the argument to this message is a unique-string, the selector for the desired method.

Similarly, to create a `CodeWindow` with a `FilePane` displaying the contents of file `f`, evaluate

```
f edit
```

A new `BrowseWindow` (described in the next section) can be created with the expression

```
user browse
```

To create a `NotifyWindow` (see Section 3) containing the error diagnostic

```
Something is wrong
```

send the message

```
user notify: 'Something is wrong'
```

The argument is a string to be displayed in the notify window's title area. (This message is used in compiled code to report error conditions encountered at execution time. It can also be used to place break pointers in a program as a debugging aide.)

To create an `InspectWindow` (see Section 3) for examining the internal state of a given object, send the message `inspect` to the object. For example, to inspect the global symbol table `Smalltalk`, evaluate

```
Smalltalk inspect
```

`To` create a new `Window` without having to specify its frame using the mouse, try

```
Window new frame: rect
```

where `rect` is an object of class `Rectangle.`

To use the framing method described above, create a window with

```
Window new newframe
```

In these last two cases, the window must be explicityly included in the window scheduler. Execute

```
user schedule: <window object>
```

If you place the cursor in a position outside any window, and press yellow bug, then the menu that appears contains several useful commands.

```
exit to overview
quit
open a subview
open a browser
open a workspace
reclaim
```

Select `open a workspace` to create a new codewindow such as that already described in Section 1. You can edit the title to reflect the purpose of the workspace.

`quit` lets you leave Smalltalk and return to the Alto Operating System.

`reclaim` attempts to recompute space consumed by various activities that seem to leave "garbage" about. It is intended for more expert users.

We will discuss the remaining commands in the section entitled Project View Windows.

EXAMPLE KINDS OF WINDOWS

Dialog Windows

Dialog windows were described in Section 1.

Code Windows An example of a code window is the *UserView workspace* which is displayed on the screen of every new Smalltalk release for your convenience, to provide access to the editing and execution facilities described in the preceding sections. Displayed in the workspace window are a number of useful Smalltalk messages which you can execute simply by selecting them and invoking the yellow-bug command `doit`. You can edit the contents of the window to include

any other messages you find yourself using frequently. If you need some blank space in which to work, simply scroll the workspace window past the end of the displayed text.

You can delete all the text to obtain a blank workspace. Do not execute the `compile` command. Doing so will cause all future copies of the workspace to be identical to the one in which you executed `compile.`

Yellow-bug commands for text editing are

```
again
copy
cut
paste
doit
compile
undo
cancel
align
```

These were described in Section 1. If you examine the class definitions, you will find that a `CodeWindow` keeps a reference to a `CodePane`, and that the yellow-bug menu, called `editmenu`, is a field of a `CodePane.`

Picture Windows

Picture windows are objects of class `BitRectEditor`. They are used for "painting" pictures on the Alto screen, and include facilities for filing and retrieving such pictures. The easiest way to create such a window is to evaluate

```
BitRect new fromuser; edit.
```

You will be given the origin and corner cursors in order to specify the rectangular area for the window. You will perceive a difference in defining the area (the rectangle you see as feedback is not a blank white one). This reflects the fact that the window is initially transparent, not opaque. Any screen information overlapping the picture window is included in the picture. This is a way of "picking up" an image that can then be stored on a disk file. To understand this use of transparency, create a picture window and then move it using the blue bug `frame` command.

The `edit` message installs a `BitRectEditor` in the window scheduler (explained in Section 5) and starts it up. The editing tools are small icons shown to the left of the picture (tool menu). These icons represent, in order, the following functions:

```
draw-thin
erase
straightedge
```

```
gray-block
paintbrush
magnifier
```

The actions for these tools are displayed above the picture (action menu). The actions are handled by the class `BitRectTool`, an object of which paints on the screen. A tool is a combination of action, mode, pen-width, gray, and grid. Action is one of

```
block-of-gray, draw, straight-edge, magnify.
```

Mode defines how the tool is combined with the current picture. It is one of

```
store, or, xor, and
```

Pen-width is the width of the drawing pen. It is either 1, 2, 4, or 8.

Gray is one of

```
black, darkgray, gray, lightgray, white
```

and grid, the minimun spacing of the mouse points, is one of 1, 2, 4, 8, 16, or 32.

To make painting work, you select a tool. A default selection of actions is given to you. Any selections you make in the action menu change the definition of the currently selected tool; these tool settings are shared by all bitRect editors. One heuristic to use would be to ignore the tools--just keep changing the actions to fill your needs.

The picture window redefines the blue-bug menu to be

```
move
grow
close
filout
printbits
```

`filout` creates a file of the bits shown on the screen. The default name for the file is `BitRect.pic`. This file can be filed into a Smalltalk system.

`printbits` creates a press file for printing named `BitRect.press` and sends the file to the printer. The file can be reprinted using `Empress.run.`

The yellow-bug does nothing.

Font Windows

Font windows are objects of class `FontWindow`. They are used for specifying the graphical representations of text characters on the screen. The best way to find out how to create and use the font editor is to use the browse window, as explained below, and selecting: the category

`Window`, the class `FontWindow`, the category `Help`, the message `help.` You will be able to execute the expressions shown there to create new fonts and use currently available ones.

Paned Windows

A paned window is a special kind of window in that it is sub-divided into areas or *panes*, each of which might contain a different kind of window. The class `PanedWindow` simply provides a way of specifying a template for the layout of the subareas and the message protocol for distributing *scheduling* and *show* requests among them. Primary among the kinds of windows used in a subarea of a `PanedWindow` are objects of the class `ListPane` or `CodePane.` A `CodePane` is a subclass of `Window`; as noted before, it handles the yellow-bug text editing menu described in Section 1. A `ListPane` is a kind of `TextFrame`; it displays a vertical list of one-line items. The list can be scrolled slow or fast, and any item can be selected. When an item is selected (or deselected) a dependent pane (subarea of the paned window) can be told to display appropriate material. Selection is done with the red mouse button (top or left button). It provides no menu for the yellow button.

The most well-known paned window is the browse window described in the next section.

THE BROWSE WINDOW

The *browse window* allows you to inspect and modify any program in the system. The browse window is a five-paned `PanedWindow`, with four `ListPanes` at the top and a larger `CodePane` below. One item at a time can be selected in each `ListPane`; as usual, the item selected is highlighted for identification. Each pane's selection controls the list of items displayed in the next pane. Together, the four list panes reflect a four-level hierarchy of all classes and their message dictionaries currently defined in the Smalltalk system.

Wake up the browse window and enter its first list pane (the one at the top left) with the mouse. The pane will show a scroll bar, which you can use to scroll its contents up or down, as described in Section 1. (Scrolling of list panes differs from that of code panes in a couple of details: the speed of scrolling is constant and independent of the cursor's vertical position within the scroll bar.) As you move from one pane to another within the window, only the pane containing the mouse cursor will display its scroll bar, to show that that is the pane currently active.

The first pane, called the *system pane*, lists broad categories of Smalltalk classes, such as 'Numbers', 'Graphical Objects', 'Windows', and so forth. Select the line `AllClasses` by pointing at it with the mouse and pressing the red button until the line is highlighted. The next pane (the *class pane*) will display an alphabetical list of all classes currently known to the system. Now try changing the selection in the system pane to, say, `Numbers`; the contents of the class pane will change to show only the classes belonging to that category (`Date, Float, Integer, LargeInteger`, etc.). For an overview of the entire system, select `SystemOrganization` in the system pane: the *code pane* (the large pane at the bottom of the browse window) will display a list of all categories of classes currently known, with the classes belonging to each. For

now, do not edit this description. An explanation of how to edit it is given in Section 3.

After selecting a category (or `AllClasses`) in the system pane, select one of its classes in the class pane, causing the third pane (the *organization pane*) to show the categories of messages defined for that class. The line `ClassDefinition` in this pane causes the code pane to display information about the structure of the class, such as its superclass and the names of its instance and class variables; selection of `ClassOrganization` results in a display of a list of the class's message categories with the messages belonging to each. When you select one of the message categories shown in the organization pane, the fourth pane (the *selector pane*) will show a list of the messages in that category. Finally, when a message is selected in the selector pane, the program for the corresponding method will appear below in the code pane.

Decompiling

Note that Smalltalk sources for all code is stored on the Woodstock server. Accessing code is done over the network. It can therefore be slow. Alternatively, the file `Smalltalk.sources` (from `Ivy<Smalltalk>`) can be placed on your disk for direct access from Smalltalk. The penalty is the large number of pages used up on your disk.

Alternatively, hold down the left shift key when selecting a method. The source code will be obtained by decompiling from the object code. As a result, no comments are displayed.

Editing

All the editing, execution, and compilation facilities discussed for `CodeWindows` are available in the code pane of the browse window. After you have edited the code for a method, the yellow-bug menu command `compile` will incorporate the newly edited version into the system, replacing the previously existing definition for that method. Once the contents of the code pane have been edited, the browse window will not allow you to make a new selection without either compiling the method as edited or restoring it to its previous state with the `cancel` command; instead, the window will signal you by flashing its code pane on the screen. (The same signal is also used if you try to `compile` or `cancel` without having made any changes in the code.)

Yellow-Bug Commands

Each pane of the browse window has its own yellow-bug menu of commands. Figure 2.1 shows the organization of the browse window and the yellow-bug menus associated with each pane. An explanation for these commands is given next.

The system pane offers the commands `filout` and `print`, both of which cause the code for the selected category of classes to be written out onto a file. (If no category is currently selected when you press the yellow button, the pane will flash and no menu will be displayed.) The difference between the two commands is that `print` generates a file in press format, with the classes in the selected category listed in alphabetical order for human consumption, whereas `filout` writes them in Bravo format in an order that reflects their subclass dependencies, so that they can later be read back into Smalltalk and reconstructed. In both cases, the name of the file

is derived from the name of the category being filed out, with embedded blanks, if any, replaced with hyphens. For example, if the category selected is 'Sets and Dictionaries', filout will write the code for that category on a file called *Sets-and-Dictionaries.st* (the extension .st stands for Smalltalk); `print` will write it on a file called *Sets-and-Dictionaries.press*. (See Section 4 for more details on printing and filing.)

On the class pane's yellow-bug menu are the commands `filout, print,` and `forget`. The first two function exactly the same way as in the system pane, except that they write out the source code for only one class, the one selected in the class pane. The name of the destination file is simply the name of the class--always one word--with the standard extension .st for `filout`, .press for `print`. The `forget` command causes the definition of the selected class to be deleted from the system. When a class is forgotten, all of its instances still in existence become *obsolete*, and will no longer respond to any messages. To give you a chance to reconsider, and to protect you from the dire consequences of invoking the forget command accidentally, the system will open a *notify window* with the message `All <classes> will become obsolete if you proceed... (where <classes>` is the name of the class being forgotten). If you are still resolved to forget the class, confirm the command by entering the notify window with the mouse and invoking the yellow-bug command `proceed` (discussed in detail in Section 3). The notify window will disappear and the class will be forgotten as requested.

The organization pane's yellow-bug menu contains the (by now familiar) commands `filout` and `print`, which work essentially the same way as in the system and class panes. As usual, the names of the files on which these commands generate their output are constructed from the name of the category selected in the organization pane, with hyphens substituted for embedded blanks and with the extension .st for `filout`, .press for `print`.

Yellow-bug commands available in the selector pane are `spawn` and `forget`. The latter is equivalent to the same command in the class pane, but causes only the selected message to be forgotten instead of a whole class. (However, since the consequences of inadvertently forgetting a message are not as grave as those of forgetting a class, there is no two-stage confirmation process--the command is simply carried out when invoked.) The `spawn` command creates a new code window -- with all the properties and capabilities of that window -- containing the current contents of the browse window's code pane (that is, the program for the message currently selected). When you issue this command, the browse window will go to sleep and the cursor will assume the form of an *origin cursor* -- you are now expected to provide the frame for the new window in the manner already described. The `spawn` command is useful in browsing or modifying sets of related messages, since it allows you to display the program for a number of messages simultaneously, each in its own window. After a `spawn`, the selector pane's selection is cleared and the code pane is reset to display the standard template for defining a new message (these templates are explained in Section 3).

PROJECT VIEW WINDOWS (subviews)

It is possible to organize your work into different projects reflected in the system by a grouping of windows. When you first resume Smalltalk, you are at the root (top view) of an n-ary tree of possible projects (see Figure 2.2). Initially, this project view tree has only the single (root) view in which you have been working. You might have already noticed a rectangular area at the top of the screen labelling this project view as `Top View`.

To create a new view, place the cursor in the area outside any window, press yellow bug and select the command `open a subview`. You will then specify a small rectangle which will serve as the entrance down a branch of the tree, to a new project view. Edit the title to give your view a unique name.

Enter this subview project window. Yellow bug has the single command `enter.` Select it.

You will enter a new screen view, initially empty but for the label at the top of the screen. This label is only edited by returning to the level containing its project view window.

To get a workspace or a browser, select the appropriate command in the yellow bug menu: `open a browser` or `open a workspace.`

To return to the next view up the tree, place the cursor outside any window and select `exit to overview.`

You can create any number of project windows at any project view level, moving down the tree by entering a project window and selecting `enter` on the yellow bug menu, and moving up the tree by selecting `exit to overview.`

You edit the name of the project by editing the title of the project window.

Remembering Changes

One significance of the project views is that each one owns its own storage (`HashSet`) named `Changes`. In it is stored pairs of class name - message selector for any such message/method you have edited or created within the context of this project. Examine it with `Changes contents sort.` To use `Changes` in filing out all changes you make, execute

```
(dp0 file: 'name of file') filout.
```

To reset storage use: `Changes init`.

To remove an item: `Changes delete: 'class selector'.`

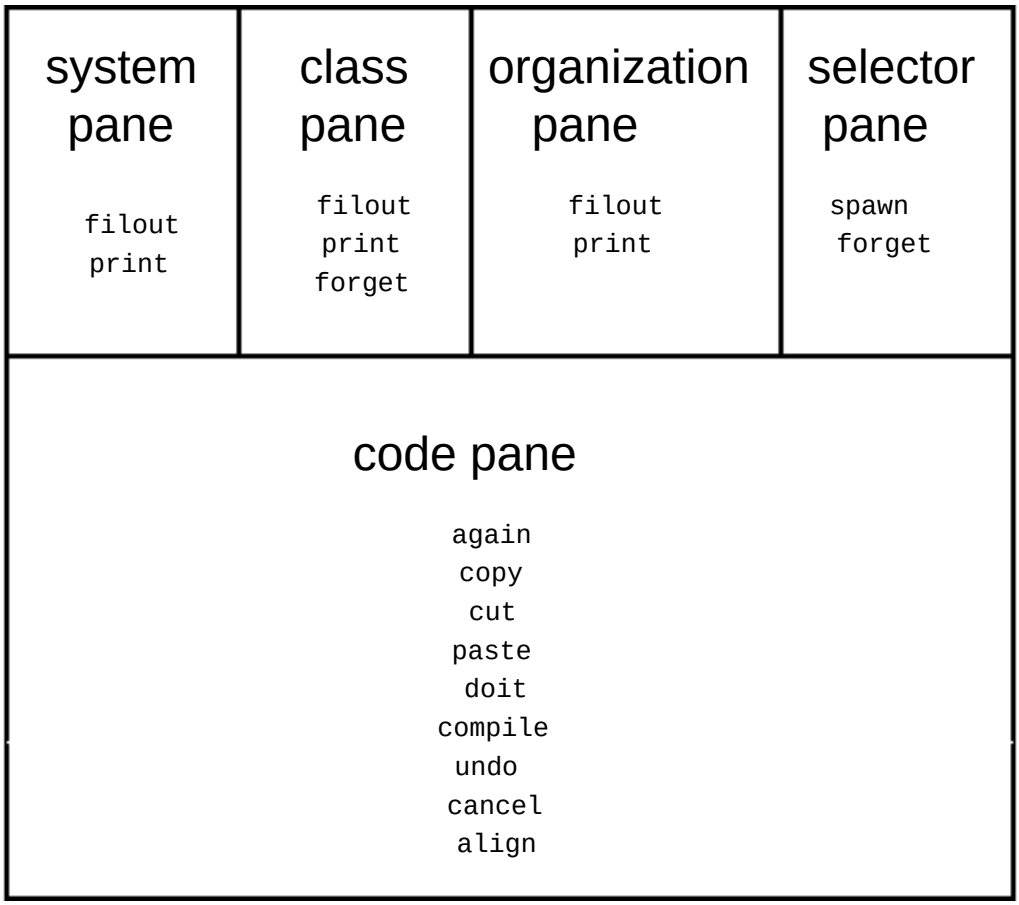

Figure 2-1. Organization and Menus for the Browse Window

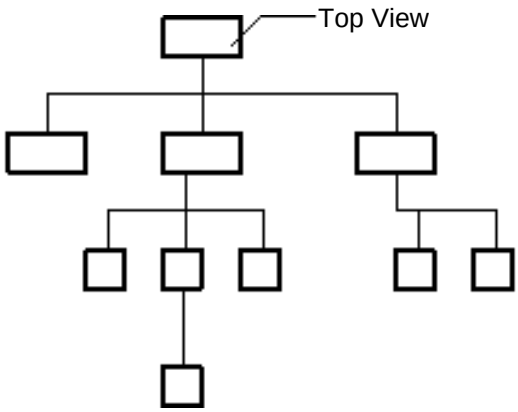

Figure 2-2a.
an n-ary tree structure
each "box" represents
a project view

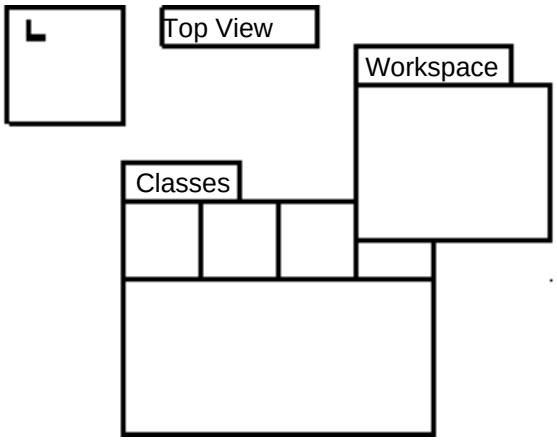

Figure 2-2b.

The initial view of
project: "Top View"

Figure 2-2. Organization of Project Views

3. Defining and Debugging Class Definitions

Defining a Class

To define a new class, select a class category in the first pane of the browse window. This selection specifies the category to which the new class will be added, and causes a template to appear in the largest pane of the browse window, the code pane. (If none of the existing categories is suitable, a new category may be added by selecting `SystemOrganization` and editing it, as described below.)

The template presented in the code pane looks as follows

```
Class new title: 'NameofClass'
subclassof: Object
fields: 'names of fields'
declare: 'names of class variables'
```

To define a new class, replace `NameofClass` by the appropriate name, leaving the quotes intact (this is a string).

If the new class is to be a subclass of a class other than just the class `Object`, replace `Object` by the desired superclass. (Note, this is not a string).

Replace `names of fields` by nothing if this class has no fields (instance variables) other than those inherited from its superclasses, or by as many field names as desired, separated by spaces (not commas!). The result is either an empty string or a string of field names.

Similarly, replace `names of class variables` by nothing if this class has no class variables, or by as many class variables as desired.

For example,

```
Class new title: 'Record'
subclassof: Dictionary
fields: 'name address telephone'
declare: ''
```

Once the editing is completed, depress the yellow mouse button and select `compile` from the menu which is presented. The new class which you have defined will be added to the designated category. When you re-select that category in the first pane of the browse window (re-selection is forced on you by the manner in which the panes are refreshed after compilation), your new class name will appear in alphabetical order in the second pane.

Defining a Category

To create a new category to which you want a new class to belong, select `SystemOrganization` in the first pane. The code pane will be filled by parenthesized expressions like the following:

```
('Kernel Classes' Class Context Object UserView VariableLengthClass )
```

The quoted part is the category name; the remaining names are of the classes which belong to that category. The order in which these parenthesized expressions appear determines the order in which the category names will appear in the first pane of the browse window. To insert a new category, simply insert a parenthesized expression whereever you want the category to appear. This expression may consist of just the quoted name of the category, or may also include the names of one or more classes already defined. Class names may appear in more than one category.

Compiling the new category list will cause your new category to be visible in the first pane. You can then select it in order to define a new class to belong to that category.

Defining a Message

To specify a new message for a class, select the class in question in the second pane of the browse window. This will cause the third pane to display

```
ClassDefinition
ClassOrganziation
'as yet unclassified'
```

if the class is newly defined.

Selecting `ClassDefinition` will present in the code pane the template which you just completed, specifying the superclass, fields, and so on. You can change this definition with varying results. For example, you can append new fields; if you insert or delete fields, then all methods will be recompiled. Selecting `ClassOrganization` will present

```
'this class has not yet been commented'
('as yet unclassified' )
```

in the code pane. The two lines of this template are invitations for further information. The first line is a request for quoted comments describing the class. These comments may be as long as you like. Especially useful is a description of how to create a member of the class. The second line is a parenthesized expression exactly like that described above for `SystemOrganization`; it invites you to classify the messages your new class will understand. The quoted part is the name of the classification; this may be followed by a list of message names that have already been defined. Typically you type none unless you are reorganizing your message categories. Also, you can have as many message categories as you find necessary.

To specify a new message, select a category, such as `'as yet unclassified'`, in the third pane. This will cause the following template to appear in the code pane

```
Message name and Arguments | temporary variables "short comment"
["long comment if necessary"
Smalltalk
Statements]
```

After editing this template to replace the message pattern, list of temporary variables (separated by spaces), comments, and method (Smalltalk statements), select `compile` in the yellow-bug menu. The message selector will appear in the fourth pane of the browse window. By browsing through the class definitions already in the system you will see many examples of this message format.

Selecting any message selector from the fourth pane will cause its program to be displayed in the code pane. The program can be modified and then compiled.

Debugging Aides: Notify and Inspect Windows

Two special kinds of windows, *notify* and *inspect* windows, provide a variety of facilities for debugging and error anaylsis in Smalltalk-76.

Notify Windows

When an error occurs in the execution of a Smalltalk program, a *notify window* will appear in the approximate center of the screen. The *title area* at the top of the window gives a brief description of the nature of the error; the contents of the window itself identify the context in which the error occurred, in the format

```
MessageClass(ReceiverClass) Selector
```

where `Selector` is the name of the Smalltalk message causing the problem, `ReceiverClass` is the class of the object to which the offending message was sent, and `MessageClass` is the class in which the method corresponding to this message is defined. Notice that `MessageClass` will always be either `ReceiverClass` itself or one of its superclasses. If the two classes are identical, the abbreviated format

```
MessageClass Selector
```

is used.

When you frame a notify window (select the `frame` command in the blue-bug menu), it opens into a `PanedWindow` with six panes. Down the left side are three `ListPanes`, which we call

stack pane

context variable pane
instance variable pane.

On the right are three corresponding `CodePanes`, the

method pane
context value pane,
instance value pane.

Yellow-bug: stack command

The yellow-bug menu for the stack pane contains the commands needed to expand the information in these six panes. Initially, the stack pane contains a single line identifying the context of the error in the format described in the preceding paragraph. The yellow-bug command `stack` in this pane expands the pane's contents into a scrollable list of contexts in the same format, representing the dynamic state of the control stack at the time of the error.

Selection in the stack pane

If you now select one of these contexts with the red button, information about that context will appear in the remaining panes of the window:

The method pane will display the Smalltalk code for the interrupted method, the context variable pane a list of arguments and method variables local to that method, and the instance variable pane a list of the instance variables (fields) of the object executing the method. Each pane can be scrolled in the usual way. You can display the value of any of the listed variables by selecting the variable name with the red button: its value will appear in the adjacent value pane. Any Smalltalk expression you evaluate (using the `doit` command) in one of the code panes will be evaluated in the stack pane's currently selected context. This often makes it possible to recover from an error by assigning new values to one or more context or instance variables and proceeding or restarting, as described below. For a closer look at the contents of an instance or context variable, select the variable name and invoke the yellow-bug command `inspect` in the variable pane--this will allow you to create an *inspect window* for examining the internal state of the object to which the variable name refers.

Yellow-bug menus

Each of the panes has its own yellow-bug menu. The stack pane menu has been described above. It contains

```
stack
spawn
proceed
restart
```

The context and instance variable panes have a menu containing only the single command `inspect`.

The remaining panes are `CodePanes`; they offer all the usual facilities for editing, executing, and compiling Smalltalk code (see Section 1).

You can inspect the code for any interrupted method simply by selecting the desired context in the stack pane. If you need to view two or more such methods at once, use the stack pane's yellow-bug menu to `spawn` a separate code window for each method. After you have diagnosed the cause of the error, you can edit and recompile the offending method or methods and continue in any of three ways:

- Select a context in the stack pane and invoke the `proceed` command on that pane's yellow-bug menu. Execution will proceed in the selected context from the point of the error.

- Select a context as above and invoke the `restart` command. The method running in that context will be reexecuted from the beginning.

- `Close` the notify window. The context of the error will be lost and you will be returned to the top level of the user interface. You can then re-issue the message that originally caused the error, or do whatever else seems appropriate in the circumstances.

In its original form, before it has been framed, a notify window consists of a single pane, corresponding to the stack pane in the window's expanded form. All the stack pane's yellow-bug commands--`stack, spawn, proceed`, and `restart`--are available when the window is in this form. Thus it is sometimes possible to save time and recover from the error without invoking the frame command.

Inspect Windows

An *inspect window* allows you to "reach inside" an object and examine or change its internal state. It consists of two panes, a list pane called the *variable pane* and a code pane called the *value pane*. The variable pane lists the names of the object's fields (instance variables): selecting one of these names with the red button causes the current contents of that field to be displayed in the value pane. (If the object being inspected belongs to a variable-length class, the variable pane will contain element numbers instead of field names. For an object with more than fifty elements, only the first twenty and the last twenty will be listed.) Any expression executed in the value pane is evaluated in the context of the object itself, so that the value of any of its fields can be set by simple assignment. The variable pane's yellow-bug menu contains the single command `inspect`, which creates a new inspect window for the object contained in the currently selected field of the original object. By repeatedly invoking this command, you can "dig," a level at a time, into an object's structure.

4. Printing and Filing out from Smalltalk

To print all the information associated with a class -- its description, the messages to which it will respond, and the methods which these messages invoke -- select the desired class in the second pane (*class pane*) of the browser and depress the yellow button to bring up a menu which will look as follows:

```
filout
print
forget
```

Selecting `print` in this menu will result in the preparation of a press file named *classname.press* where *classname* is the name of the selected class. When the printer is unknown or not available, a message appears in the system *Dispframe* and a menu of possible printers appears at the center of the screen. The printer which you select the very first time will be your default printer, e.g. `MENLO`. Subsequent selections do not change this default. If the menu appears again, select the same printer, or a different printer, or `none`, which appears as the last option. The screen will go black during this process. The press file will be left on your disk at the end of this transaction.

Selecting `filout` in the menu will result in the preparation of a Bravo format file named *classname.st* which is written on your disk. This file may be printed by transferring it to a Bravo disk, if you desire, but its main usefulness is that it may be filed into a new Smalltalk system (the press file cannot). To this end, it is better to print the class on the same file as its superclasses using the `print` command in the first (*system*) pane.

Categories of classes (in the first, *system*, pane of the browser window) also respond to `print` and `filout`. In this case, all classes categorized in the selected category will be printed or filed out. The difference between the two commands is that `print` generates a file in `press` format, with the classes in the selected category listed in alphabetical order for human consumption, whereas `filout` writes them in Bravo format in an order that reflects their subclass dependencies, so that they can later be read back into Smalltalk and reconstructed. In both cases, the name of the file is derived from the name of the category being filed out, with embedded blanks, if any, replaced with hyphens. For example, if the category selected is `'Sets and Dictionaries'`, `filout` will write the code for that category on a file called *Sets-and-Dictionaries.st* (the extension .st stands for Smalltalk); `print` will write it on a file called *Sets-and-Dictionaries.press*

The *organization* pane's yellow-bug menu contains the (by now familiar) commands `filout` and `print`, which work essentially the same way as in the *system* and *class* panes. As usual, the names of the files on which these commands generate their output are constructed from the name of the category selected in the organization pane, with hyphens substituted for embedded blanks and with the extension .st for `filout`, .press for `print`.

Press files which are no longer needed may be deleted from within Smalltalk by executing the expression

```
(dp0 file:   'classname.press') delete
```

If you wish press files to be deleted automatically after printing, you may modify the standard Smalltalk release as follows. For classes, modify class `Class` by changing the method for the message `printout` to read:

```
dp0 delete:  title + '.press'
```

For categories of classes, modify class `SystemOrganizer` by changing the method for the message `printCategory` to read:

```
dp0 delete:  (cat + '.press') as FileName
```

To send `printout` to a printer different from the default, modify class `Pressfile` by changing the method for the message `toPrinter` from

```
self toPrinter: PrinterName
```

to

```
self toPrinter: 'Clover'
```
(or other name of printer)

Alternatively, change the value of the object `PrinterName`

```
PrinterName _ 'Menlo'.
```

If your chosen printer is not available (e.g., there is a time out in the attempt to transmit the information), then the system will inform you of the situation and present a menu of all the printers from which you can select a new choice for this transmission only. You might choose to retry the same printer.

5. Smalltalk Objects

SUBCLASS STRUCTURE

- Object
  - Array
    - CoreLocs
    - Interval
    - Paragraph
      - TextEntity
    - String
      - UniqueString
      - Natural
    - Substring
    - Vector
  - BitBlt
  - BitRectTool
  - Class
    - VariableLengthClass
  - ClassOrganizer
    - SystemOrganizer
  - Context
    - RemoteContext
  - Cursor
    - Decompiler
  - Dict
    - File
      - AltoFile
      - WoodstockFile
      - JuniperFileController
    - FileDirectory
      - AltoFileDirectory
      - FtpDirectory
      - JuniperInterface
      - WoodstockFileDirectory
    - FilePage
      - AltoFilePage
      - EtherFilePage
        - WoodstockFilePage
        - JuniperPageBuffer
  - DictionaryEntry
  - Etherworld
  - ExceptionHandler
  - FieldReference
  - FontWindow
  - Form
  - FormSet
  - Generator
  - HalfToner
  - HashSet
    - Dictionary
      - SymbolTable
    - MessageDict
  - JuniperParameterBlock
    - JuniperRequestParameterBlock

- JuniperResultParameterBlock
- Menu
- MessageTally
- Number
  - Date
  - Float
  - Int32
  - Integer
  - LargeInteger
  - MachineDouble
- ObjectReference
- Pacbuf
- ParagraphPrinter
  - BravoPrinter
  - PressPrinter
- ParagraphScanner
- ParsedAssignment
- ParsedConditional
- ParsedConjunct
- ParsedDisjunct
- ParsedFieldReference
- ParsedLoop
- ParsedMessage
  - ParsedNegation
- ParsedObjectReference
- ParsedRemote
- Parser
- ParseStack
- Point
  - UserEvent
- PressFile
- PriorityInterrupt
- PriorityScheduler
- RadioButtons
- Reader
- Rectangle
  - BitRect
- RemoteParagraph
- ScrollBar
- Socket
  - RetransmitSocket
    - NameUser
    - RPPSocket
      - EFTPSender
    - WSocket
    - JuniperSocket
  - RoutingUpdater
- Stream
  - Dispframe
  - FileStream
  - ParsedBlock
  - PQueue
    - SafeQ
  - Queue
    - EventQueue
  - Set
    - Image
      - BitImage

- Document
- Heading
- Path
- SetReader

Textframe
- ListPane
  - ClassPane
  - OrganizationPane
  - SelectorPane
  - StackPane
  - SystemPane
  - VariablePane
- TextImage
  - BorderedText
  - ParagraphEditor

TextStyle
Time
Timer
TokenCollector
- FieldNameCollector

Turtle
- PressTurtle

UserView
VirtualMemory
Vmapper
WidthTable
Window
- BitRectEditor
- CodePane
- DocumentEditor
- FilePane
- PanedWindow
  - BrowseWindow
  - CodeWindow
  - InspectWindow
  - NotifyWindow
  - ProjectWindow
  - SyntaxWindow

SMALLTALK GLOBAL OBJECTS

Globals you should know about

BitBlt Colors

background	Integer
black	Integer
dkgray	Integer
gray	Integer
ltgray	Integer
white	Integer

BitBlt Modes

erasing	Integer
oring	Integer
storing	Integer
xoring	Integer

Cursor Shapes

CornerCursor	Cursor
DownCursor	Cursor
NormalCursor	Cursor
OriginCursor	Cursor
ReadCursor	Cursor
UpCursor	Cursor
WaitCursor	Cursor
XeqCursor	Cursor

File Directories

dp0	AltoFileDirectory
dp1	AltoFileDirectory
dpw	WoodstockFileDirectory
dpj	JuniperInterface

User interface

Changes	HashSet
user	UserView
NotifyFlag	Object
Undeclared	SymbolTable

Miscellaneous

DefaultTextStyle	TextStyle
mem	CoreLocs
Smalltalk	SymbolTable
spy	MessageTally
Top	PriorityScheduler
PrinterName	String

Globals you may run into

User interface

AllClassNames	Vector	
defaultBitRectEditor	BitRectEditor	
Events	EventQueue	
kbMap	String	
sysFontWindow	FontWindow	
SystemOrganization	SystemOrganizer	

Miscellaneous

IntervalFrom1By1	Interval	1 to: 32767 by: 1
nullString	String	
PressScale	Integer	32
UpperCase	String	

Globals you should never run into

Pools

AltoFilePool	SymbolTable
ByteCodes	SymbolTable
FilePool	SymbolTable
TokenCodes	SymbolTable
WoodstockFilePool	SymbolTable
EtherPool	SymbolTable
JuniperConstants	SymbolTable

Virtual Memory

BitMasks	SymbolTable
FirstContext	Context
Flushed	Object
Pmap	VirtualMemory
SpecialOops	Vector
Vmem	VirtualMemory

Compiler

FilinSource	Object
Huh	String
HuhFlag	Object
MethodKeeper	Stream
MethodKeeperKeeper	Object
UST1	Vector
USTable	Vector
WhatFlag	Object

Miscellaenous

Counter	String
FontDict	Dictionary
ThePicture	Object
Xlate	String
Xlated	Vector

6. Scheduling Round-Robin Fashion

The job of managing the Smalltalk window interface -- deciding which window is awake and relaying information to that window about user actions with the mouse, keyboard, and keyset -- is one of the services provided by the ubiquitous special object `user`. In one of its fields, called `sched, user` maintains a list of all windows and other kinds of objects currently active on the display screen.

A window `w` can be brought into existence by executing the expression

```
w _ Window new newframe.
```

This will supply an origin cursor with which you can frame the window (as described in Section 2). The window `w` can then be placed on the list `sched` by executing **one** of the following:

```
user schedule:  w
user scheduleOnBottom:  w
user restartup:  w
```

The first expression places `w` at the front of the `sched`, the second places it at the back. The third is the same as the first except that Smalltalk returns to the top level (as in CTRL-SHIFT-ESC or `user restart`), the cursor is forced into the window `w`, and the window is immediately awakened.

To remove (the first occurrence of) `w` from `sched`, execute the expression:

```
user unschedule:  w
```

The order in which windows are listed in the list `sched` determines their relative "depth" as they appear to you on the screen: objects listed earlier in `sched` appear nearer to the "front" of the display, and may overlap partially or completely hide those further back in `sched.` Whenever a window wakes up, it is "promoted" to the beginning of `sched` (and therefore to the front of the screen), and all windows previously ahead of it are moved back one position to make room.

To "live" in `sched`, an object must be of a "schedulable type". To qualify as schedulable, it must understand three messages: `firsttime, eachtime`, and `lasttime`, which mean, approximately, "Do you want to wake up?" "Do you want to remain awake?" and "Go to sleep". Methods for responding to these messages are defined in class `Window`, so any object belonging to a subclass of `Window` -- such as `BrowseWindow, CodeWindow`, or `NotifyWindow` -- will automatically understand them. (The subclass may, of course, override `Window's` methods with definitions of its own.) An object in `sched` needn't actually be a window, as long as it knows how to behave like one by responding to these three messages. The "dialog window," for example, belongs to class `Dispframe`, which is not a subclass of `Window;` but since it understands the messages `firsttime, eachtime,` and `lasttime, it` is perfectly at home in `sched.`

The message `lasttime` should return `self` if it wants the scheduler to select the next window to awaken by scanning `sched` in the normal top-down order. It should return `false` if it wants the

scheduler to scan `sched` in bottom-up (reverse) order. The latter is appropriate if the window got put to sleep by the user's invocation of the UNDER command in the blue button menu.

HOW THE SCHEDULER WORKS

The information above should be enough for you to get going with scheduling windows. If you have difficulties using the scheduler, you may want to read the rest of this documentation, which describes its operation in more detail.

A context installed in Smalltalk's top-level priority scheduler drives the window interface by repeatedly sending the message

```
user run
```

`user's` method for responding to this message is as follows (paraphrasing slightly for simplicity):

```
run | i w
   [while  true do
      [i _ 0.
      until   ( (i _ i + 1) > sched length or
               (w _ sched   i) firsttime) do  [].
      i > sched length   []
      sched promote   w.
      while  w eachtime do  [].
      w lasttime]]
```

This method consists of a single outer loop, to be repeated as long as true is true -- in other words, forever. On each pass through this outer loop, the inner `until` loop scans through the list `sched`, sending each window (or other object) the message `firsttime`, meaning "Do you want to wake up?" The answer is up to the window itself to decide, usually by asking its screen `frame` (an object of class `Rectangle`) whether it contains the current location of the mouse cursor. (Notice that, since `sched` is scanned from front to back, the window that wakes up will be the foremost window containing the cursor.) However the window arrives at its decision, it will respond to the message `firsttime` with the value `false` if it wants to remain asleep, `true` (that is, any value other than false) if it wants to wake up. In addition, if the answer is `true`, the window will perform any actions it considers appropriate upon awakening, such as repainting itself on the screen (overlaying any other windows appearing in front of it), highlighting a selection, growing a scroll bar, or whatever else a particular flavor of window may require.

The scan through `sched` continues until either the end of `sched` is reached or one of the windows answers `true` to the message `firsttime`, meaning "Yes, I want to wake up." In the former case, the next line of the method says to do nothing and repeat the outer loop, restarting the scan at the beginning of `sched`. If, on the other hand, the scan ended because window `w` asked to wake up, `user` asks its `sched` to promote `w` to the front, then enters another loop that says "Keep sending `w` the message `eachtime`, and do nothing as long as the answer is `true`." The window will answer `eachtime` with `true` or `false`, indicating whether it wants to remain awake or go to sleep;

if it chooses to remain awake, it will also perform any chores associated with doing its thing," such as interrogating the mouse and keyboard and responding as appropriate. Thus, although the inner `while` loop of the method above has a null body, the repeated execution of the message

```
w eachtime
```

as its termination test causes the window to transact its normal business. When the window decides to go to sleep (for example, if a mouse button is pressed outside its frame), the loop terminates and the next line is executed, sending the window the mesage `lasttime.` This instructs it to close up shop and go to sleep, and it will comply after its fashion. The outer loop of the `user run` method will then be repeated, initiating another scan through `sched`.

The description just given of `user's` window-scheduling algorithm is accurate in its gross outlines, but in real life matters are complicated by some wrinkles. For example, the `run` message is actually `run:`, and takes an argument, called `topFlag`. A non-false argument value means that the window on the front of `sched` is already awake, and that `sched` should not be scanned until this window decides to go to sleep.