Archives
Columns
Features
Print Archives 1994-1998

Special
BYTE Digest
Michael Abrash's *Graphics Programming Black Book*
101 Perl Articles

About Us
How to Access BYTE.com
Write to BYTE.com
Advertise with BYTE.com

ARTICLES

15 Years Ago in BYTE

Are languages being used according to their original design? Larry Tesler of Apple thought they should be....ideally.

BYTE devoted almost the entire issue to Smalltalk, Smalltalk-80, and other object-oriented-programming topics. In one article, Apple's Larr Tesler said, "You can write almost any program better in a language you know well than in one you know poorly. But if languages are compared from a viewpoint broader than that of a narrow expert, eac language stands out above the others when used for the purpose for which it was designed." Hold that thought.

August 1981 's almost-exclusive coverage of Smalltalk provided a wealth of information from programming and debugging, to the graphics kernel. Here's what else Larry Tesler had to say:

The Smalltalk Environment

by Larry Tesler

[Editor's Note: Due to the large volume of graphics originally included with this article, you will need to refer to the print issue, page 90, to view them.]

As I write this article, I am wearing a T-shirt given to me by a friend.

Emblazoned across the chest is the loud plea:

DON'T

MODE

ME IN

Surrounding the caption is a ring of barbed wire that symbolizes the trapped feeling I often experience when my computer is "in a mode."

In small print around the shirt are the names of some modes I have known and deplored since the early 1960s when I came out of the darkness of punched cards into the dawn of interactive terminals. My rogues' gallery of inhuman factors includes command modes like INSERT, REPLACE, DELETE, and SEARCH, as well as that inescapable prompt, "FILE NAME?" The color of the silk scr een is, appropriately enough, very blue.

My friend gave me the shirt to make fun of a near-fanatical campaign have waged for several years, a campaign to eliminate modes from th face of the earth--or at least from the face of my computer's display screen. It started in 1973 when I began work at the Xerox Palo Alto Research Center (PARC) on the design of interactive systems to be used by office workers for document preparation. My observations of secretaries learning to use the text editors of that era soon convinced me that my beloved computers were, in fact, unfriendly monsters, an that their sharpest fangs were the ever-present modes. The most common question asked by new users, at least as often as "How do I do this?," was "How do I get out of this mode?" Other researchers have also condemned the prevalence of modes in interactive systems for novice users.

Novices are not the only victims of modes. Experts often type commands used in one mode when they are in another, leading to undesired and distressing consequences. In many systems, typing the letter "D" can have meanings as diverse as "replace the selected character by D," "insert a D before the selected character," or "delete the selected character." How many times have you heard or said, "Oops, I was in the wrong mode"?

Preemption

Even when you remember what mode you are in, you can still fall into a trap. If you are running a data-plotting program, the only command you can use are the ones provided in that program. You can't use any of the useful capabilities of your computer that the author of the program didn't consider, such as obtaining a list of the files on the disk. On the other hand, if you're using a program that lets you list files, you probably can't use the text editor to change their names. Also, if you are using a text editor, you probably can't plot a graph from the numbers that appear in the document.

If you stop any program and start another, data displayed by the first program is probably er ased from the screen and irretrievably lost fro view. In general, "running a program" in most systems puts you into mode where the facilities of other programs are unavailable to you. Dan Swinehart calls this the *dilemma of preemption* .

Many systems feature hierarchies of modes. A portion of a typical mode hierarchy is shown in figure 1. If you are in the editor and want to copy text from a file, you issue the *copy-from* command and it give the prompt "from what file?" You then type a file name. What if you can't remember the spelling? No problem. Leave *from-what-file* mode leave *copy-from* mode, *save* the edited text, *exit* from the editor to th executive, call up *file management* from the executive, issue the *list-files* command, look for the name you want (Hey, that went by too fast. Sorry, you can't scroll backwards in that mode.), *terminate* the list command, *exit* from file management to the executive, reenter th *editor* , issue the *copy-from* command, and when it prompts you with "from-what-file?," simply type the name (you haven't forgotten it, have you?).

You don't have to be a user-sympathizer to join the campaign against modes. The most coldhearted programmer is a victim as well. Say you have programmed a new video game for your personal computer and have encountered a bug. An obscure error message appears on the screen mixed in with spacecraft and alien forms. To see the latest version of the program on the screen, you have to wipe out the very evidence you need to solve the problem. Why? Because the system forces you to choose between edit mode and execute mode. You can't have both.

Enter the Integrated Environment

Soon after I began battling the mode monster, I became associated with Alan Kay, who had just founded the Learning Research Group (LRG) at the Xerox PARC. Kay shared my disdain for modes and had devised a user-interface paradigm (reference 3) that elimin ated one kind of mode, the kind causing the preemption dilemma. The paradigm he advocated was called "overlapping windows."

Most people who have used computer displays are familiar with windows. They are rectangular divisions of the screen, each displaying a different information set. In some windowing systems, you can have several tasks in progress, each represented in a different window, and can switch freely between tasks by switching between windows.

The trouble with most windowing systems is that the windows compet with each other for screen space-if you make one window bigger, another window gets smaller. Kay's idea was to allow the windows to overlap. The screen is portrayed as the surface of a desk, and the windows as overlapping sheets of paper. Partly covered sheets peek out from behind sheets that obscure them. With the aid of a pointing device that moves a cursor around the screen, you can move the

cursor over a partly covered sheet and press a button on the pointing device to uncover t hat sheet.

The advantages of the overlapping-window paradigm are:

-- the displays associated with several user tasks can be viewed simultaneously

-- switching between tasks is done with the press of a button

-- no information is lost switching between tasks

-- screen space is used economically

Of course, windows are, in a sense, modes in sheep's clothing. They are more friendly than modes because you can't slip into a window unknowingly when you are not looking at the screen, and because you can get in and out of any window at any time you choose by the push of a button.

Kay saw his paradigm as the basis for what he called an "integrated environment." When you have an integrated environment, the distinction between operating system and application fades. Every capability of your personal computer is always available to you to apply to any information you want. With minimal effort, you can move among such divers e activities as debugging programs, editing prose, drawing pictures, playing music, and running simulations. Information generated by one activity can be fed to other activities, either by direct user interaction or under program control.

When Kay invented the Smalltalk language in 1972, he designed it with the ability to support an integrated environment. The implementations of Smalltalk produced by Dan Ingalls and the other members of the Learning Research Group have achieved ever-increasing integration. The file system, process-management system, graphics capability, and compiler are implemented almost entirely in Smalltalk. They are accessible from any program, as well as by direct user interaction.

In recent years, the idea of an integrated environment has spread outside the Learning Research Group and even to non-Smalltalk systems. The window-per-program paradigm is now commonplace, and many system designers have adopted the overlapping-sheet model of the screen.

In summary, the term *environment* is used to refer to everything in a computer that a person can directly access and utilize in a unified and coordinated manner. In an *integrated environment* , a person can interweave activities without losing accumulated information and without giving up capabilities.

Strengths of Smalltalk

Before delving further into the *nature* of the Smalltalk environment, w
should first discuss its *purpose* .

Many general-purpose programming languages are more suitable for certain jobs than others. BASIC is easy to learn and is ideal for small dialogue-oriented programs. FORTRAN is well suited to numerical applications. COBOL is tailored to business data processing. Pascal is good for teaching structured programming.

LISP is wonderful for processing symbolic information. APL excels at manipulating vectors and matrices. C is great for systems programming. SIMULA shines at discrete simulations. FORTH lets people quickly develop efficient modular p rograms on very small computers.

All these languages have been used for numerous purposes in additio to those mentioned. You can write almost any program better in a language you know well than in one you know poorly. But if language are compared from a viewpoint broader than that of a narrow expert, each language stands out above the others when used for the purpos for which it was designed.

Although Smalltalk has been used for many different applications, it excels at a certain style of software development on a certain type of machine. The machine that best matches Smalltalk's strengths is a personal computer with a high-resolution display, a keyboard, and a pointing device such as a *mouse* or graphics tablet. A cursor on the screen tracks mouse movements on the table so you can point to objects on the screen. The mouse (reference 4) has one or more buttons on its top side. One button is used as a *selection button* If there are more buttons, they are normally used as *menu b uttons* .

If the machine has a high-performance disk drive, you can use a virtual-memory version of Smalltalk and have as little as 80 K bytes o main memory, not counting display-refresh memory. Otherwise, you should have at least 256 K bytes of memory. This much memory is required because the whole integrated environment lives in one address space. It includes not only the usual run-time language support, but window-oriented graphics, the editor, the compiler, and other software-development aids. The programs you write tend to be small because they can build on existing facilities; no system facilities are hidden from the user. Users of LISP and FORTH will be familiar with this idea.

Smalltalk supports its preferred hardware by incorporating software packages that provide:

-- output to the user through overlapping windows

-- input from a keyboard, a pointing device, and menus

-- uniform treatment of textual, graphical, symbolic, and numeric information

These interactive facilities are utilized heavily by the built-in programming aids and are available to all userwritten applications.

The style of software development to which Smalltalk is oriented is *exploratory* . In exploratory development, it should be fast to create and test prototypes, and it should be easy to change them without costly repercussions. Smalltalk is helpful because:

-- The language is more concise than most, so less time is spent at th keyboard.

-- The text editor is simple, modeless, and requires a minimum of keystrokes.

-- The user can move among programming, compiling, testing, and debugging activities with the push of a button.

-- Any desired information about the program or its execution is accessible in seconds with minimal effort.

-- The compiler can translate and relink a single change into the environment in a few seconds, so the time usually wasted waiting for recompilation after a small program modification is avoided.

-- Smalltalk programs grow gracefully. In most environments, a system gets more difficult to change as it grows. If you add 2 megabytes of virtual memory to the Smalltalk environment, you can fill the second megabyte with useful capabilities as fast as you can fill the first.

-- The class structure of the language prevents objects from making too many assumptions about the internal behavior of other objects (see David Robson's article, "Object-Oriented Software Systems," on page 74 of this issue). The programmer can augment or change the methods used in one part of a program without having to reprogram other parts.

The Anatomy of a Window

Over the years, members of the Learning Research Group have embellished Kay's original window concept. Let us look at a Smalltalk window in more detail (figure 2).

The window is shown as a *framed* rectangular area wi th a *title tab* attached to its top edge. The program associated with the window must confine its output to the framed area.

Every window has a *window menu* . The window menu includes

commands to reframe the window in a new size and location, to close the window, to print the contents of the window on a hard-copy device, and to retrieve windows hidden under it.

A window is tiled by one or more *panes* , each with its own pane menu. The pane menu includes commands appropriate to the content of that pane. In addition, a pane has a *scroll bar* on its left side used t scroll the contents of the pane when more information exists than fits in the frame at one time.

Although you can see many windows and panes at once, you can interact with only one pane at a time. That pane and its window are said to be awake or *active* . To awaken a different pane of the same window, move the cursor over the new pane. To awaken a different window, move the cursor over the new win dow and press the *selection button* on the pointing device. When a window wakes up, its title tab and all its panes are displayed, and it is no longer covered up by other windows.

The scroll bar of the active pane is called the *active scroll bar* . Its menu and the menu of its window are called the *active menus* . In order to reduce screen clutter and maximize utilization of precious screen space, no inactive scroll bars or menus are displayed. On machines that use a pointing device with three buttons, some version of Smalltalk even hide the active menus until one of two menu button is pressed, at which time the associated menu pops up and stays up until the button is released. If the button is released when the cursor over a command in the menu, that command is executed.

Modeless Editing

The overlapping-window paradigm helps eliminate preemption. It can also reduce the need for certain prompts and their associated modes. For example, you never have to type the name of a procedure you want to examine. At worst, you point to its name in a list; at best, the desired procedure is already in a window on the screen, and you activate that window.

Unfortunately, overlapping windows do not eliminate command modes like "insert" and "replace" by themselves. Between 1973 and 1975, I worked at PARC with various collaborators, including Dan Swinehart and Timothy Mott, to banish command modes from interactive systems. Despite initial skepticism, nearly all users of our prototypes grew to appreciate the absence of modes. The following techniques were devised by us to eliminate modes from text editing. They are analogous to the techniques used to keep Polish-notation calculators relatively mode-free. Similar techniques can be applied to page layout graphics creation, and other interactive tasks.

Selection precedes command:

-- Every command is executed immediately when you issue it. You are

not asked to confirm it. Yo u can issue an undo command to reverse the effects of the last issued command. Although the main purpose of "undo" is to compensate for the lack of command confirmation, it can also be used to change your mind after issuing a command.

-- For a command like "close the active window" that requires no additional parameters, you simply issue the command.

-- For a command like "delete text" that requires one parameter, you first select the parameter using the pointing device and then issue the command. Until you issue the command, you can change your mind and make a different selection, or even choose a different command.

-- For a command like "send electronic mail" that requires several parameters (recipient, subject, content), you first fill the parameters into a form using modeless text editing and then issue the command. You are not in a mode while filling out the form. If you want to copy something into the form from another place, you can. If you want to do something else instead, just do it; you may even return to the forr later and finish filling it out.

Typing text always replaces the selected characters:

-- Pressing a text key on the keyboard never issues a command. It always replaces the current selection by the typed character and automatically selects the gap following that character.

-- To replace a passage of text, first select it and then type the replacement. The first keystroke deletes the original text.

-- To insert between characters, you first select the gap between thos characters and then type the insertion. Essentially, you are replacing nothing with something.

-- The destructive backspace function always deletes the character preceding the selection, even if that character was there before the selection was made.

-- The "undo" command can be used to reverse the effects of all your typing and backspacin g since you last made a selection with the pointing device.

Thus, the usual insert, append, and replace modes are folded into one mode-replace mode-and one mode is no mode at all.

The "shift lock" key and analogous commands like "bold shift" and "underline shift" cause modes for the interpretation of subsequently typed characters. However, shifts are familiar to people and are relatively harmless. The worst they do is change a "d" to a "D," "d," o "d"never to a Delete command.

The bit-map display can show boldface characters, as well as italics, underlining, and a variety of styles and sizes of printer's type. Thus, a you enter text in bold shift, the screen shows what the text will look like when it is printed. A command like bold shift can also be applied t existing text to change it to boldface.

In 1976, Dan Ingalls devised a user interface for Smalltalk that incorporated most of the mode-avoidance techniques discussed earlie Consequently, it is rare in the present Smalltalk environ ment to encounter a mode.

Making a Selection

In the Smalltalk-76 user interface, text is selected using the pointing device and a single button. First, the cursor is moved to one end of th passage to be selected. The selection button is pressed and held dowr while the cursor is moved to the other end of the passage. This operation is called "draw-through," though it is not necessary to traverse intermediate characters en route to the destination. When th cursor reaches the other end of the passage, the button is released. The selected passage is then shown in inverse video.

The feedback given to the user during selection is as follows. When th button is depressed, a vertical bar appears in the nearest intercharacter gap. (At the left end of a line of text, the bar appears t the left of the first character. At the right end of a line, the bar appeai to the left of the final space character.)

If the button is released without moving the cursor, the bar remains, indicatin g that a zero-width selection has been made. This method-clicking once between characters-is the one to use before you insert new text.

If the button is held down while the cursor is moved, the system supplies continuous feedback by highlighting in inverse video all characters between the initial bar and the gap nearest to the cursor. When the button is released, the selected characters remain highlighted. This method-drawing through a passage is the one to use before you copy, move, delete, or replace text, or before you change to boldface or otherwise alter its appearance.

Clicking the button twice with the cursor in the same spot within a word selects that whole word and highlights it. This special mechanisr is provided because it is very common to select a word. Informal experiments lead us to believe that double clicking is much easier tha drawing through a word for beginners and experts alike. It is also faster. It takes the average user about 2.6 seconds to select a word anywhere on the s creen using draw-through, but it takes only 1.5 seconds using the double click (reference 5).

There is only one selection in the active pane. It is called the *active selection* .

Issuing a Command

When you issue a command in Smalltalk, you are sending a message to an object. There are two ways to send a message from Ingalls's user interface. You can send certain commonly sent messages to the active pane or window by choosing them from menus; you can send any message to any object by direct execution of a Smalltalk statement.

Smalltalk-76 provides pop-up menus for the most commonly used commands, like "cut," which deletes the selected text. To issue the "cut" command, you pop up the active-pane menu with one of the menu buttons on the mouse, keep that button down while moving the cursor to the command name, and then release the button. A command in the pane menu can have only one parameter, the active selection. A command in the window menu can have no parameters.

To issue a command that is not available in a menu, you select any place you can insert text, and type the whole command as a statemer in the Smalltalk language. Then you select that statement and issue the single-parameter command "do it" to obtain the result. The "do it' command provides immediate execution of any Smalltalk statement o group of statements. This method of command issuance uses the previous method: you are sending the message `doit` to the pane, with the Smalltalk statement as its parameter.

It is standard practice to keep a "work-space" window around the screen in which to type your nonmenu commands. When you want to reissue a nonmenu command issued earlier, simply select the command in the workspace window and "do it." You may, of course, edit some of the parameters of the old command before you select it and "do it." In a sense, you are filling out a form when you edit parameters of an immediate statement.

Unfortunately, the common commands "move text from here to ther e and "copy text from here to there" cannot be issued by a single menu command because they require two parameters, the source selection and the destination selection. Sometimes, they even involve message to more than one pane, the source pane and the destination pane. In modeless system, a move or copy command is done in two steps:

-- A move is done by *cut and paste* . First, you select the source text and issue the "cut" command. The "cut" command deletes the selecte text, but leaves it in a special place where it can be retrieved by "paste." Then you select the destination and issue the "paste" command to complete the move.

-- A copy is done by *copy and paste* , which is completely analogous t cut and paste, but does not delete the original text.

Remember the "copy-from-file" example (the one where you had to g in and out of many layers of modes)? In the Smalltalk-76 user

interface, you can accomplish this with six pushed buttons, no mode ex its, and no typing: (1) activate the source window that displays the file you are copying from; (2) select the desired text; (3) issue the "copy" command in the menu; (4) activate the destination window; (5 select the destination point, and (6) issue the "paste" command in the menu. The job requires little more effort than copying within the same document. If the window is not already on the screen and you can't remember the file name, you can go to another window and scroll through a list of files without having to exit any modes, invoke any programs, save any edits, lose sight of the destination file, or lose any time.

The Smalltalk-76 text-editing facilities not only relieve you of the burden of modes, they also require very few keystrokes and are easy to learn.

Software-Development Aids

One of my summer projects in 1977 was to increase the speed and friendliness of the Smalltalk software-development environment by adding *inspect windows* , *browse windows* , an d *notify windows* to the user interface. These and other enhancements made by the Learning Research Group are described below. In recent months, the team has further enhanced the Smalltalk-80 environment. Although it conforms to the same principles as before, its details are different from what is described in this article.

Inspecting Data Structures

Suppose someone has given you a Smalltalk program to implement a "regular polygon" class and you want to learn more about it. It would be helpful to see an actual instance of a regular polygon.

If the variable `triangle` refers to a regular polygon, you type the following statement into your work-space window:

```
triangle inspect
```

and then issue the "do it" command in the pane menu. In a few seconds, a two-paned "inspect window" appears on the screen. Its titl tab tells you the class of the inspected object, in this case, RegularPolygon. The window is divided into two panes. The left or variable pane lists the parts of a regular polygon, sides, center, radius and plotter. The right or value pane is blank.

You point to the word sides in the variable pane and click the selection button on the mouse. The word sides is highlighted, and in the value pane, the value of the variable sides appears, in this case, 3. You poir to the word center and click. In the value pane appears the value of center, in this case, the point 526@302. The value pane is dependent on the variable pane because its contents are determined by what you

select in the variable pane. The arrow in figure 3 symbolizes this dependency.

Let's inspect the value of center. In the variable pane, where center is selected, pop up the pane menu and issue the "inspect" command. Or the screen appears another inspect window showing that center is an instance of class `Point` . You can now examine that point's variables, x and y, reactivate the original inspect window, close either or both windows, or work in any other window. You are no t in a mode.

Browsing Through Existing Definitions

Now that you have inspected a sample regular polygon, you might want to find out what methods have been defined in its class. One wa to do this is to activate a window called a "browse window" or "browser." Most Smalltalk programmers leave a browser or two on the screen at all times with the work-space window.

The title tab of the browser says "Classes" because the standard browser lets you examine and change the definitions of all Smalltalk classes-classes supplied by the system, as well as classes supplied by yourself. It is easy to create a more restricted browser that protects the system from ill-conceived modification. But on a personal computer, you are just going to hurt yourself.

The browser has five panes. The principal dependencies between panes are symbolized by arrows in figure 4. The top row has four panes called the class-category pane, class pane, method-category pane, and method pane. The large lower pane i s called the editing pane. (After you have used the system for a few minutes, the significance of each pane becomes apparent, and it is not necessary t memorize their technical names.)

In photo 13a, the browser shows a method definition in the editing pane. You can tell that the method is class RegularPolygon's version o scale: because RegularPolygon is highlighted in the class pane and scale: is highlighted in the method pane.

The method-category pane lists several groups of methods within clas `RegularPolygon` : initialization, analysis, display, transformation, testing, and private methods. You can tell that scale: is a transformation message in class `RegularPolygon` because that category is highlighted.

The class-category pane lists several groups of classes, including numbers, files, and graphical objects. You can tell that class `RegularPolygon` is in the graphical objects group because that category is highlighted.

Suppose you want to look at a different meth od, `translateBy:` . Click it name in the method pane and its definition is immediately displayed i that pane's dependent, the editing pane. If the method you want to see is in the method category analysis, first click that category name.

Immediately after you do that, its dependent, the method pane, lists the methods in that category. Now you can click the name of the desired method.

If you want to know things about the class as a whole, like its superclass and field names, click "Class Definition" in the method-category pane and the definition appears in the editing pane.

Suppose you want to look at a different class, say `IrregularPolygon` . Click its name in the class pane and its method categories are immediately displayed in the next pane. If the class you want to see is in the class category windows, first click that category name. Immediately after you do that, the class pane lists the classes in that category. Now you can click the name of the desired class.

Categorizati on is used at both the class and method level to help the programmer organize his or her program and to provide fewer choices in each pane. If a list is longer than what can fit in a pane, it can be scrolled by pressing a mouse button with the cursor in the scroll bar.

If you just want to browse around reading class and method definitions, you can do so by lazily clicking the selection button with the cursor over each name, never touching the keyboard. That is why the window is called a browser. Browsers are further discussed in references 6 and 7.

Astute readers may have noticed that the class template (see "The Smalltalk-80 System" by the Learning Research Group on page 36 of this issue) presents the methods of a class apart from the methods of its instances, while the browser does not. This discrepancy stems from differences between the Smalltalk-80 and Smalltalk-76 languages.

Revising Definitions

If you are looking at a method definition or class definition in the editing pane, you can revise it using the standard text-editing facilities (select, type, cut, paste, copy).

If you like, you can copy information into the definition from other windows--including other browse windows--because you are not in an mode while browsing. You can even interrupt your editing to run another program, list your disk files, draw a picture, or do whatever you like. You can later reactivate the browser and continue editing.

When you are done editing, pop up the active-pane menu and issue the "compile" command. Compilation takes a few seconds or less because it is incremental-that is, you can compile one method at a time. The compiler reports a syntax error to you by inserting a message at the point where the error was detected and automatically selecting that error message. You can then cut out or overtype the message, make the correction, and immediately reissue the "compile" command.

If you start to revise a definition and change your mind about it, you can pop up the pane men u and issue the "cancel" command. The "cancel" command redisplays the last successfully compiled version of the method. If you cancel by accident, just issue the "undo" commanc to return the revised version.

Adding New Definitions

To add a new method definition, select a method category. In the editing pane, a template appears for defining a new method. The template reminds you of the required syntax of a method.

Use standard editing facilities to supply the message pattern, variable list, and body of the method. When the definition is ready, issue the "compile" command.

Once compilation succeeds, the selector of the new method is automatically added to the alphabetized list in the method pane, and the message pattern is automatically changed to boldface in the editing pane.

A new class definition is added in an analogous manner. Start by selecting a class category, then fill in a template for defining a new class and compile it. New categories can be added and old cat egories can be renamed and reorganized.

Program Testing

Let us purposely add a bug to a method and see how it can be tracke down and fixed.

Browse to the method `cornerAngle` in class `RegularPolygon` , cut out the characters "180-", and recompile it. In the `RegularPolygon` work-space window, select the test program and issue the "do it" command. Instead of the desired triangle, an open three-sided figure is drawn because of the bug introduced into the angle calculation.

Breakpoints

To track down the bug, let us set a breakpoint in the method `cornerAngle` . Using standard editing facilities, add the statement:

```
self notify: 'about to calculate angle'.
```

before the return statement. Now rerun the test case. When the computer encounters the breakpoint, a new window appears in midscreen. It is called a "notify window". The title tab of the notify window says "about to calculate angle".

The notify window ha s one pane, the stack pane. It shows `RegularPolygon > > cornerAngle` (ie: the class and method in which the

breakpoint was encountered). The pop-up menu of that pane offers several commands, including "stack" and "proceed".

The "proceed" command closes the notify window and continues execution from the breakpoint. If we issue a "proceed" in our example the same breakpoint will be encountered again immediately because the cornerAngle method is used several times during the execution of the test program.

What a Notify Window Can Display

The "stack" command expands the contents of the pane to include messages that have been sent, but have not yet received replies. It reveals that the sender of the message cornerAngle was `RegularPolygon > >plot:`.

The pop-up menu of the notify window offers the usual repertoire, including the "close" and "frame" commands. If "close were issued, the notify window would disappear from the screen and execution of the program under test would be aborted. Let us issue the "frame" command instead. The notify window grows larger and acquires a total of six panes. Their interdependencies are diagrammed in figure 5.

The upper left pane is the stack pane retained from before. The upper right pane is an editing pane. If you select `RegularPolygon > > plot:` in the stack pane, its method definition appears in the editing pane. You can scroll through the definition and even edit it there and recompile as in the browser.

The middle two panes are the "context variable" and "context value" panes. They are analogous to the two panes of an inspect window, but in this case, the variables you can examine are the arguments and local variables of the method selected in the stack pane. Click ink in the variable pane to see its value in the value pane.

The bottom two panes are the "instance variable" and "instance value" panes. They also are analogous to the panes of an inspect window. They let you examine the instance variables of the receiver of the message selected in the stack pane. Click center to see its value appear in the value pane.

You can type statements into the value panes and execute them using "do it". They will be executed in the context of the method selected in the stack pane-that is, they may refer to arguments and local variables of the method and to instance variables.

Debugging

You could step through the execution of the method in the editing pane. You would select one statement at a time in the editing pane and issue the "do it" command. To close in on the planted bug, we can evaluate self cornerAngle, an expression on the last line of the

method. Select that expression and issue the "do it" command. The answer, 1 20, appears to the right of the question. Since the interior angle of a regular triangle is 60 degrees, we have found the planted bug.

Now select `RegularPolygon` > > `cornerAngle` in the stack pane. Its method definition, including the breakpoint we set, appears in the editing pan Use standard editing to remove the breakpoint, correct the error, and recompile the editing pane.

You can randomly access any level in the stack by clicking it in the stack pane.

Resumption

After recompiling a method, you can resume execution from the beginning of any method on the stack using the "restart" command in the stack-pane menu. This lets the test proceed without having to sta over from the work-space window. Resumption of execution after a correction is a handy capability when a program that has been runnin well encounters a minor bug.

The entire stack of the process under test was saved in the notify window. When a notify window appears, the rest of the system is not preempted. You are not required to deal with the notify window when it appears. You can work in other windows and come back to it later, cause other notify windows to be created, or work a little in the notify window and then do somet hing else. There are no modes.

Error Notifications

Error messages are no different from breakpoints, except that if they are supposed to be "unrecoverable" they are programmed as:

```
self error: 'error whatever'.
```

If the user "proceeds" out of the notify window after an error, the process under test is terminated.

The most frequently encountered Smalltalk error is "Message not understood." It occurs when a method is sent to an object and neithe that object's class nor any of its superclasses defines a method to receive that message. Let us edit the method sideLength to send the message cosine instead of cos. After recompiling that method and reexecuting the test program, a notify window appears to announce that class Real and its superclasses do not define cosine.

In most programming systems, equivalent error conditions such as "undeclared procedure" and "wrong number of arguments" are issued at compile time. Smalltalk cannot detect these conditions until r un time because variables are not declared as to type. At run time, the

object sent the message cosine could be an instance of a class that di
define a method of that name.

Type Checking

When we program in languages like Pascal, we depend on type checking to catch procedure-call errors early in the software-development process. In return, we have to take extra time maintaining type declarations, and we lose the very powerful ability to define generic or "polymorphic" procedures with the same name but with parameters of varying types.

Type checking is important in most systems for four reasons, none of which is very important in Smalltalk:

-- Without type checking, a program in most languages can "crash" in mysterious ways at run time. Even with type checking, most programming systems can crash due to uninitialized variables, dangling references, etc. Languages with this feature are sometimes called "unsafe." Examples of unsafe languages are Pascal, PL/1, and C Examples of fairly safe languages are BASIC and LISP. Smalltalk is a safe language. It cannot be wiped out by normal programming. In particular, it never crashes when there are "type mismatches." It just reports a "Message not understood" error and helps the programmer quickly find and fix the problem through the notify window.

-- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable. In Smalltalk, type errors are found early in testing, along with value-range errors and other bugs.

-- Type declarations help to document programs. This is true, but wel chosen variable names and pertinent comments provide more specific information than do type declarations. A poor documenter can convey as little information in a strongly typed program as in an untyped program.

-- Most compilers can generate more efficient object code if types are declared. Existing implementations of Smalltalk cannot take advantag e of type declarations. We expect that future versions will have that ability. At that time, type declarations may be added to the language. They probably will be supplied by the system rather than the user, using a program-analysis technique called "type inference."

Project Windows

Although overlapping windows enable you to keep the state of several tasks on the screen at the same time, you may sometimes be working on several entirely different projects, each involving several tasks. Smalltalk lets you have a different "desk top" for each project. On eac desk top are windows for the tasks involved in that project. To help you travel from one desk top to another, a desk top can have one or more project windows that show you other available desk tops and let

you switch to one of them.

Saving Programs

In unintegrated systems, you create a program using standard text-editing facilities. Then, using standard utility programs, you can obtai a program listing on paper, back up the program on other media, and transmit the program to other people. In an integrated system, equivalent capabilities must be provided within the system itself. Som of the program-saving capabilities of Smalltalk are described briefly below.

One important facility is the snapshot. The entire state of the Smalltal environment-including class and method definitions, data objects, suspended processes, windows on the screen, and project desk tops-can be momentarily frozen and saved on secondary storage. The snapshot can be restored later and resumed. People familiar with the sysout in InterLISP or the workspace concept in APL will understand the benefit of this facility.

Another facility allows definitions of one or more methods or classes t be listed on a printer. A related facility is `filin/filout` . The filout message writes an ASCII representation of one or more definitions onto a conventional text file. The definitions can then be transfused into another Smalltalk environ ment by using the filin message in that environment.

Often, during a programming session, the user changes a number of method definitions that are scattered throughout many classes and cannot recall which ones were changed. The changes facility automatically keeps a record of what definitions changed in each project, and makes it easy for the user to filout those definitions at th end of the session.

Implementation of the Environment

Because Smalltalk is an integrated environment, all the facilities described in this article are implemented in the high-level language, including modeless editing, windows, the compiler, and the notify mechanism. This was possible because Smalltalk represents everything, including the dynamic state of its own processes, as objects that remember their own state and that can be sent messages by other objects. Using the browser, you can examine and (carefully) change the definitions of the software-development aids.

In the implementation of Smal ltalk-76, classes `InspectWindow` , `BrowseWindow` , and `NotifyWindow` are all tiny subclasses of class `PanedWindow` which defines their common behavior. Similarly, classes `StackPane` , `VariablePane` , `ValuePane` and so on, are all tiny subclasses of class `ListPane` . The superclass defines common behavior such as scrolling and selecting entries.

If someone shows you a system claimed to be "Smalltalk," find out whether the software-development aids exist and whether they are programmed as class definitions in the high-level language. If not, the system is not bona fide.

Conclusions

The Smalltalk programming environment is reactive. That is, the user tells it what to do and it reacts, instead of the other way around. To enable the user to switch between tasks, the state of the tasks is preserved in instantly accessible windows that overlap on desk tops. To give the user the maximum freedom of choice at every moment, modes rarely occur in the user interface. The result of this organizatio is that tasks, including softwaredevelopment tasks, can be accomplished with greater speed and less frustration than is usually encountered in computer systems.

References

1. Sneeringer, J. *"User-Interface Design for Text Editing: A Case Study."* Software-Practice and Experience 8, pages 543 thru 557, 1978.

2. Swinehart, D C (thesis). *"Copilot: A Multiple Process Approach to Interactive Programming Systems."* Stanford Artificial Intelligence Laboratory Memo AIM-230, Stanford University, July 1974.

3. Kay, A and A Goldberg. *"Personal Dynamic Media."* Computer, March 1977 (originally published as Xerox PARC Technical Report SSL 76-1, March 1976, out of print).

4. English, W, D Engelbart, and M Berman. *"Display-Selection Techniques for Text Manipulation."* IEEE Transactions on Human Factors in Electronics, volume 8, number 1, pages 21 thru 31, 1977.

5. Card, S, T Moran, and A Newell. *"The Keystroke-Level Model for User Performance Time with Interactive Systems."* Communications o the ACM, volume 23, number 7, July 1980.

6. Goldberg, A and D Robson. *"A Metaphor for User-Interface Design."* Proceedings of the Twelfth Hawaii International Conference on System Sciences, volume 6, number 1, pages 148 thru 157, 1979.

7. Borning, A. *"ThingLab-A Constraint-Oriented Simulation Laboratory."* To appear in ACM Transactions on Programming Languages and Systems (originally published as Stanford Computer Science Report STAN-CS-79-746 and Xerox PARC Technical Report SSL-79-3, July 1979, out of print).

August 1981

photo_link (125 Kbytes) 

Larry Tesler, Apple Computer, Inc., 10260 Bandley Dr., Cupertino, CA 95014.