BYTE®

the small system

AUGUST 1981 Vol. 6, No. 8
$2.50 in USA/$2.95 in Canada

A McGraw-Hill Publication

In The Queue

Volume 6, Number 8

August 1981

Features

14 **Introducing the Smalltalk-80 System** by Adele Goldberg / A readers' guide to the Smalltalk articles in this issue.

36 **The Smalltalk-80 System** by the Xerox Learning Research Group / How message-sending objects are used in the Smalltalk-80 system.

50 **Build a Z8-Based Control Computer with BASIC, Part 2** by Steve Ciarcia / Steve continues his description of the Z8-BASIC Microcomputer and suggests two applications.

74 **Object-Oriented Software Systems** by David Robson / Object-oriented software systems provide the underlying design of Smalltalk.

90 **The Smalltalk Environment** by Larry Tesler / Programming and debugging in Smalltalk are always interactive activities.

147 **User-Oriented Descriptions of Smalltalk Systems** by Trygve M H Reenskaug / A Smalltalk application program will limit the user's access to the language.

168 **The Smalltalk Graphics Kernel** by Daniel H H Ingalls / The Graphics Kernel provides the interface through which all text and graphics are displayed.

200 **The Japanese Computer Invasion** by Stan Miastkowski / Like it or not, the Japanese small computers are on their way.

230 **Building Data Structures in the Smalltalk-80 System** by James C Althoff Jr / Many kinds of data structures can be added easily to the Smalltalk-80 system.

286 **Design Principles Behind Smalltalk** by Daniel H H Ingalls / The design principles of a language strongly affect its power and usability.

300 **The Smalltalk-80 Virtual Machine** by Glenn Krasner / The use of a Smalltalk-80 Virtual Machine allows the system to be transported easily among different 16-bit microprocessors.

322 **Building Control Structures in the Smalltalk-80 System** by L Peter Deutsch / Design of complicated control structures is easy in the Smalltalk-80 language.

348 **Is the Smalltalk-80 System for Children?** by Adele Goldberg and Joan Ross / Although Smalltalk-80 is not meant to be used by children, application programs can be written that will allow them to be creative and, at the same time, learn about programming.

369 **ToolBox: A Smalltalk Illustration System** by William Bowman and Bob Flegal / The versatile Smalltalk-80 language can create an environment for graphics design that can be used by non technically oriented people.

378 **Virtual Memory for an Object-Oriented Language** by Ted Kaehler / Virtual memory techniques must be used when the active memory space needed by a language is much larger than the amount of available memory.

Reviews

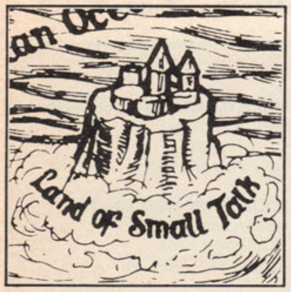
398 Microsoft Editor/Assembler Plus by Keith Carlson

401 BOSS: A Debugging Utility for the TRS-80 Model I by Scott Mitchell

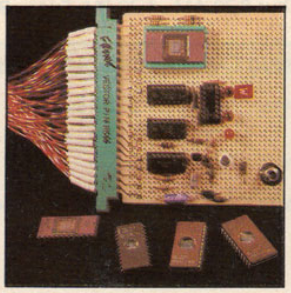
Nucleus

6 Editorial: Smalltalk: A Language for the 1980s
30 Letters
197 BYTE's Bits
224 BYTELINES
392 BYTE's Bugs
388 Ask BYTE
391 Books Received
391 Software Received
392 Clubs and Newsletters
394 Event Queue
402, 413 System Notes: Indirect I/O Addressing on the 8080; Aim-65 16-bit Hexadecimal to Decimal Conversion
404, 408, 414, 417, 418 Programming Quickies: A Disk Catalog for the Eighties; Alpha-Beta Tree Search Converted to Assembler; Fast Line-Drawing Techniques; Word Ujbnmarle; Binary-to-BCD Converter Program for the 8080
421 What's New?
478 Unclassified Ads
479 Reader Service
480 BOMB, BOMB Results

BYTE



Page 14



Page 50

Page 90

Page 168

Editorial

Smalltalk: A Language for the 1980s

by Chris Morgan, Editor in Chief

Welcome to the fifth annual BYTE language issue. Over the past four years we have devoted our August issues to discussions of APL, Pascal, LISP, and FORTH, respectively. This year we are pleased to present the Smalltalk-80 language, the culmination of ten years of research by the Xerox Learning Research Group located at the Xerox Palo Alto Research Center (PARC) in California.

During the past few months the BYTE staff has been acquainting itself with Smalltalk. I spent some time this spring working with the Smalltalk systems at Xerox PARC and being briefed by Adele Goldberg and Dave Robson. I came away excited by this revolutionary language. I hope the articles in this issue convey some of that excitement.

Smalltalk is an *object-oriented* language, as opposed to *procedure-oriented* languages such as BASIC, Pascal, and FORTRAN. Because of this, programming in Smalltalk is similar to the process of human interaction. An analogy might help to clarify this point. Suppose a person wishes to invest in a good mutual fund. He sends a telegram to his broker. The broker analyzes the current state of the market and picks what he considers to be the best mutual fund for his client. That in a very small nutshell describes the basic activity inherent in all Smalltalk programs: a message is sent to a receiver to invoke some response. In our analogy, the telegram is the message and the broker is the receiver. The telegram has two parts, called the *selector* and the *argument*. Here the selector is "buy" and the argument is "best mutual fund." The broker belongs to a *class* which contains the description of the *method* he uses to pick the best mutual fund. Because of this, the client does not have to tell the broker how to do his job.

Of course, my analogy skims only the thinnest surface of the deep waters of the Smalltalk-80 system, as you'll see when you read the articles in this issue.

When I first worked at a Smalltalk-80 computer terminal, I noticed an interesting phenomenon: I did very little typing, although a full keyboard was available to me. This is because of the window menu format and the presence of the "mouse," a small mechanical box with wheels that lets you quickly move the cursor around the screen. (Stoney Ballard of Digital Equipment Corporation, who has been doing research work lately with the Smalltalk-80 system, points out that he was able to do a significant amount of programming with his experimental system over several weeks even though his keyboard was not working.) Choosing a particular item in a list from a window causes another window to appear on the screen. Additional levels of nested windows can be accessed by continuing to reposition the cursor and pressing the appropriate key on the mouse.

This makes for *fast* programming. Those who saw the remarkable demonstration of the Xerox Star terminal (Xerox's new $16,000 office terminal) at the National Computer Conference (NCC) this past spring got a taste of what a programming environment can do for productivity.

Smalltalk allows the user to solve more problems without becoming a computer expert. Larry Tesler from Apple (who wrote "The Smalltalk Environ-

Smalltalk-80 is a trademark of Xerox Corporation.

ment" on page 90) spoke about the efficiency of the language at the NCC. For example, suppose a user is running a complex program that churns away for nearly an hour—then a bug appears in the output routine. All is not lost. Since the Smalltalk-80 language is "modeless" (a concept Tesler discusses in his article), the user can debug the output routine and continue with the main routine without having to start from the beginning. This is only one of the advantages of the Smalltalk-80 system.

Where to Start

The order in which you read the Smalltalk-80 articles in this issue makes a difference. The first stopping point should be Adele Goldberg's article "Introducing the Smalltalk-80 System" on page 14, in which she provides a guided tour of the issue. I also recommend Dave Robson's "Object-Oriented Software Systems" on page 74 as a good overview of the Smalltalk-80 philosophy. The glossary on page 48 will be helpful as you begin to absorb the rather extensive (and sometimes overwhelming) vocabulary used to describe the language. I found that, once the terms become familiar, the concepts begin to make elegant sense.

When Can I Buy It?

There are currently no personal computer implementations of the Smalltalk-80 language. Because of this, I'm sure we'll be criticized by some for introducing the language too early and frustrating our readers. Nevertheless, I feel that the time to begin exposing people to object-oriented language is *now*. Only by challenging and enticing the personal computer community can we stimulate the industry to create the machines we all dream of.

As far as future hardware hopes are concerned, it is interesting to note that four of the speakers at the recent NCC Smalltalk-80 symposium were from Digital Equipment Corporation, Apple Computer Company, Tektronix, and Hewlett-Packard. All four research representatives were quick to point out that their companies are not necessarily working on Smalltalk products, but are rather exploring the language's potential. Despite the disclaimers, though, I would be very surprised if we do not see a computer with the Smalltalk-80 system built in sometime in the next few years—perhaps sooner. I hope this issue brings that dream closer.■

Acknowledgments

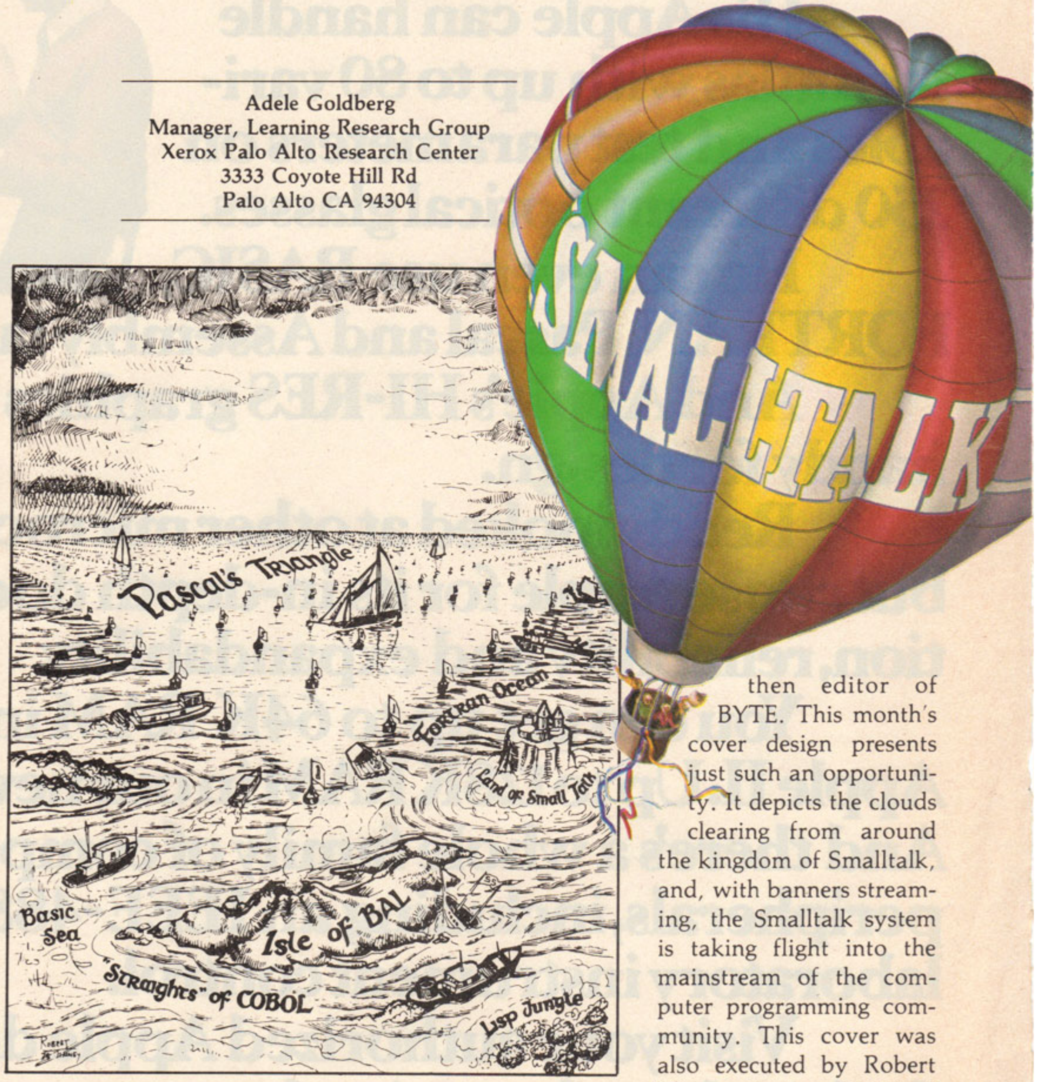
I wish to express my appreciation to Adele Goldberg and Dave Robson at Xerox PARC for their invaluable help in preparing this special issue—and especially to Adele for coordinating the many authors who contributed their expertise. I'd also like to thank Gregg Williams for his editorial skills in preparing this issue . . . . CM

Introducing the Smalltalk-80 System

Adele Goldberg
Manager, Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

It is rare when one can indulge in one's prejudices with relative impunity, poking a bit of good humored fun to make a point.

With this statement, Carl Helmers opened his remarks in the "About the Cover" section of the August 1978 issue of BYTE. The issue was a special on the language Pascal, so Helmers took the opportunity to present Pascal's triangle as drawn by artist Robert Tinney. The primary allegory of the cover was the inversion of the Bermuda Triangle myth to show smooth waters within the area labeled "Pascal's Triangle." In explaining the allegory, Helmers guided the traveler through the FORTRAN Ocean, the BASIC Sea, around the Isle of BAL, and up to the Land of Smalltalk.



Traveling upward (in the picture) through heavy seas we come to the pinnacle, a snow white island rising like an ivory tower out of the surrounding shark infested waters. Here we find the fantastic kingdom of Smalltalk, where great and magical things happen. But alas . . . the craggy aloofness of the kingdom of Smalltalk keeps it out of the mainstream of things.

It is rare when one can indulge in one's fantasies to respond to so pointed a remark as that provided by the then editor of BYTE. This month's cover design presents just such an opportunity. It depicts the clouds clearing from around the kingdom of Smalltalk, and, with banners streaming, the Smalltalk system is taking flight into the mainstream of the computer programming community. This cover was also executed by Robert Tinney, to the delight of the Learning Research Group (LRG) of the Xerox Palo Alto Research Center. LRG is the group that has designed, implemented, and evaluated several generations of Smalltalk over the past ten years.

The balloon on the cover symbolizes the Smalltalk-80 system that is being released this year for more general access. The release is in the form of publications and a file containing the Smalltalk-80 programming system. Twelve articles describing the system appear in this issue of BYTE. Through such publication, LRG's research will become generally accessible, dispelling the clouds.

Smalltalk is the name LRG assigned to the software

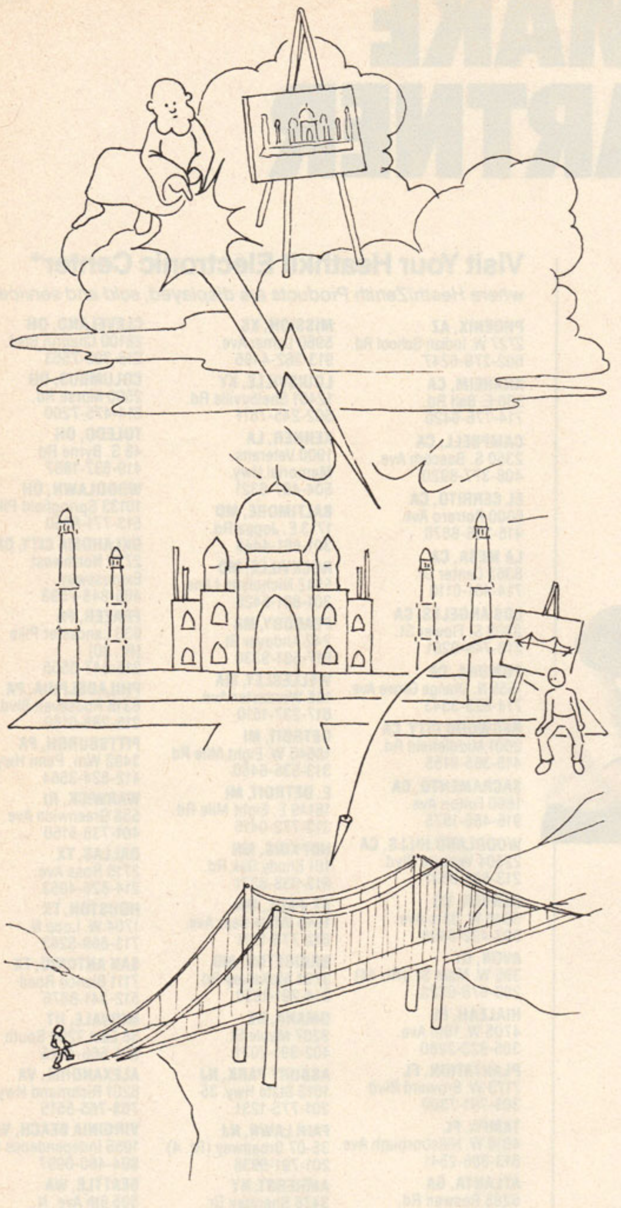


Figure 1

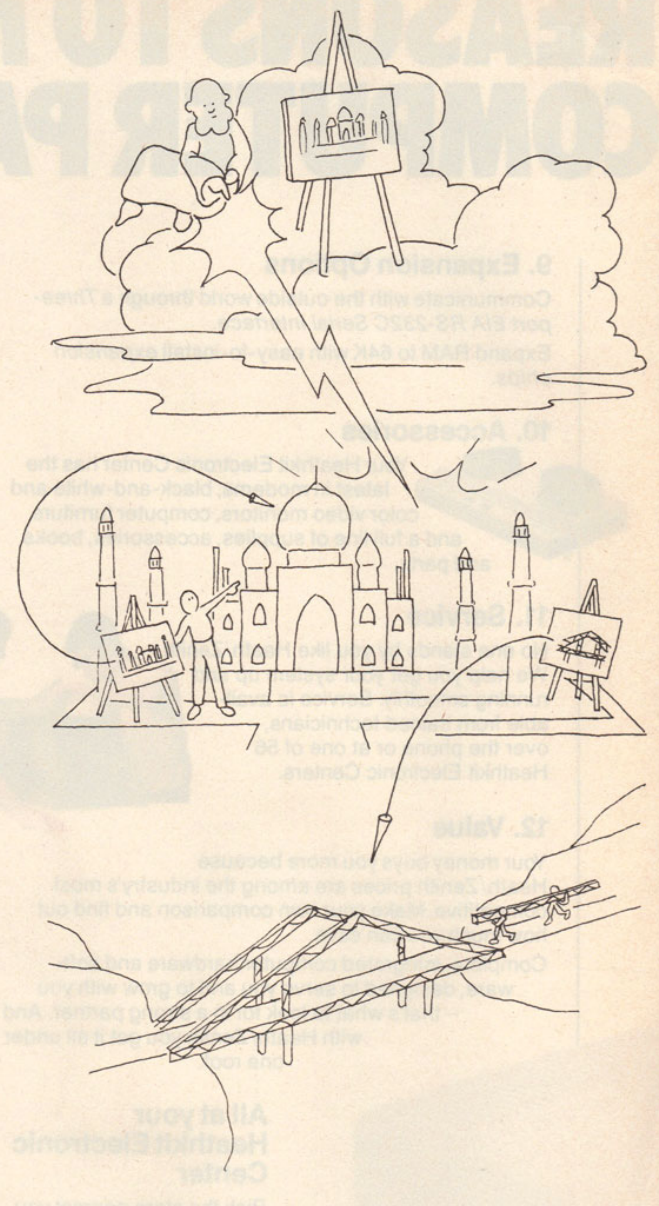


Figure 2

part of Alan Kay's personal computing vision, the Dynabook. The vision is a hand-held, high-performance computer with a high-resolution display, input and output devices supporting visual and audio communication paths, and network connections to shared information resources. LRG's goal is to support an individual's ability to use the Dynabook creatively. This requires an understanding of the interactions among language, knowledge, and communication. To this end, LRG does research on the design and implementation of programming languages, programming systems, data bases, virtual memories, and user interfaces.

The ivory tower on the island of Smalltalk is an exciting, creative place in which to work on these ideas. A sense of LRG's long-range goals is aptly portrayed in the illustrations designed by Ted Kaehler.

In figure 1, we see a view of the conventional software development environment: a wizard sitting on his own computational cloud creating his notion of a Taj Mahal in which programmers can indulge in building applications for nonprogramming users. The Taj Mahal represents a complete programming environment, which includes the tools for developing programs as well as the language in which the programs are written. The users must walk whatever bridge the programmer builds.

A goal in the design of the Smalltalk system was to create the Taj Mahal so that programmers can modify it by building *application kits*, which are specialized exten-

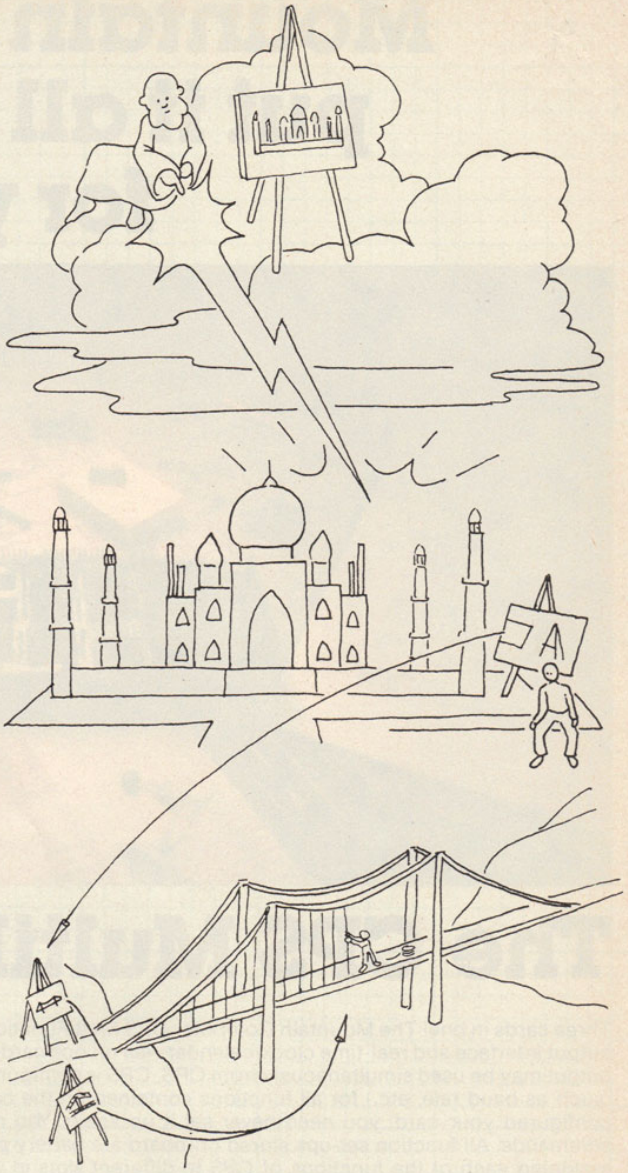


Figure 3

sions and/or subsets of the system whose parts can be used by a nonprogrammer to build a customized version of the application. Applications that can be created from a kit are related in a fundamental way: the programmer may, for example, create it for building bridges, but it is the user who pieces together the parts to create a customized bridge (see figure 2).

One of LRG's current research goals is to provide system parts to aid the programmer in creating kits. Although Smalltalk itself is conceptually sufficient for this task, it needs better support to help the programmer piece together the graphical display and the control for an interactive user interface. This is the "kit maker," as shown in figure 3.

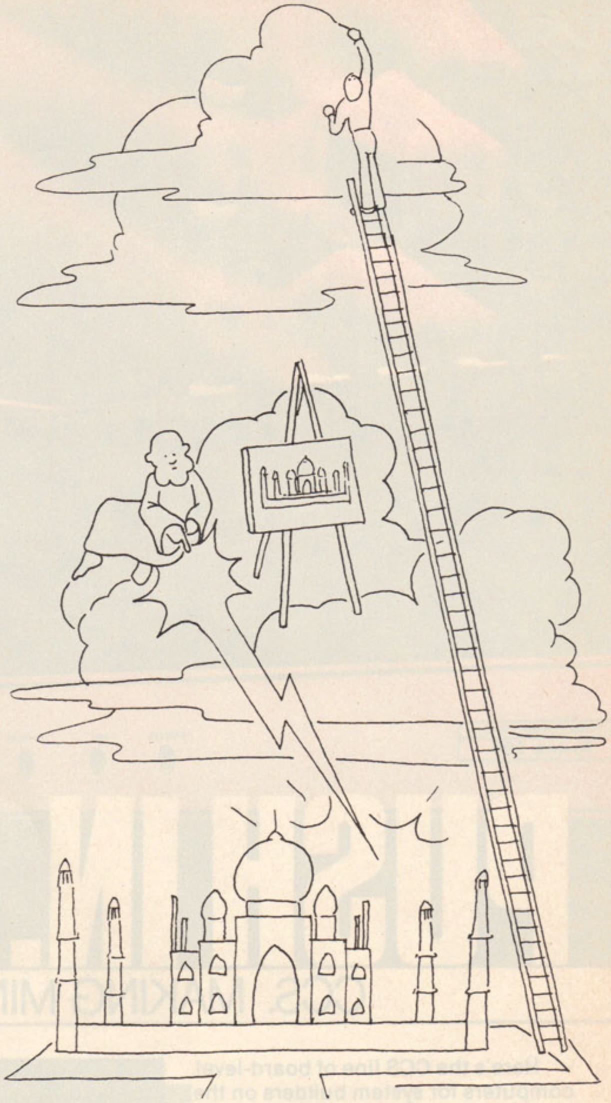


Figure 4

As part of the Dynabook vision, the system should help the programmer build a personal computational cloud (see figure 4). Two research projects, ThingLab by Alan Borning and PIE by Ira Goldstein and Danny Bobrow, took advantage of Smalltalk's support for creating new metaphors.

We are often asked: "What makes Smalltalk different from other languages?" The articles in this issue attempt to answer that question. Look for an emphasis on interactive graphics, on modular development of programs, and on integrated approaches to accessing program development tools. Also, look for the distinction between a programming language and a programming system, and consider the difference in providing a system in which the user can feel individual mastery over complexity. Although each article can be read independently of the

Text continued on page 26

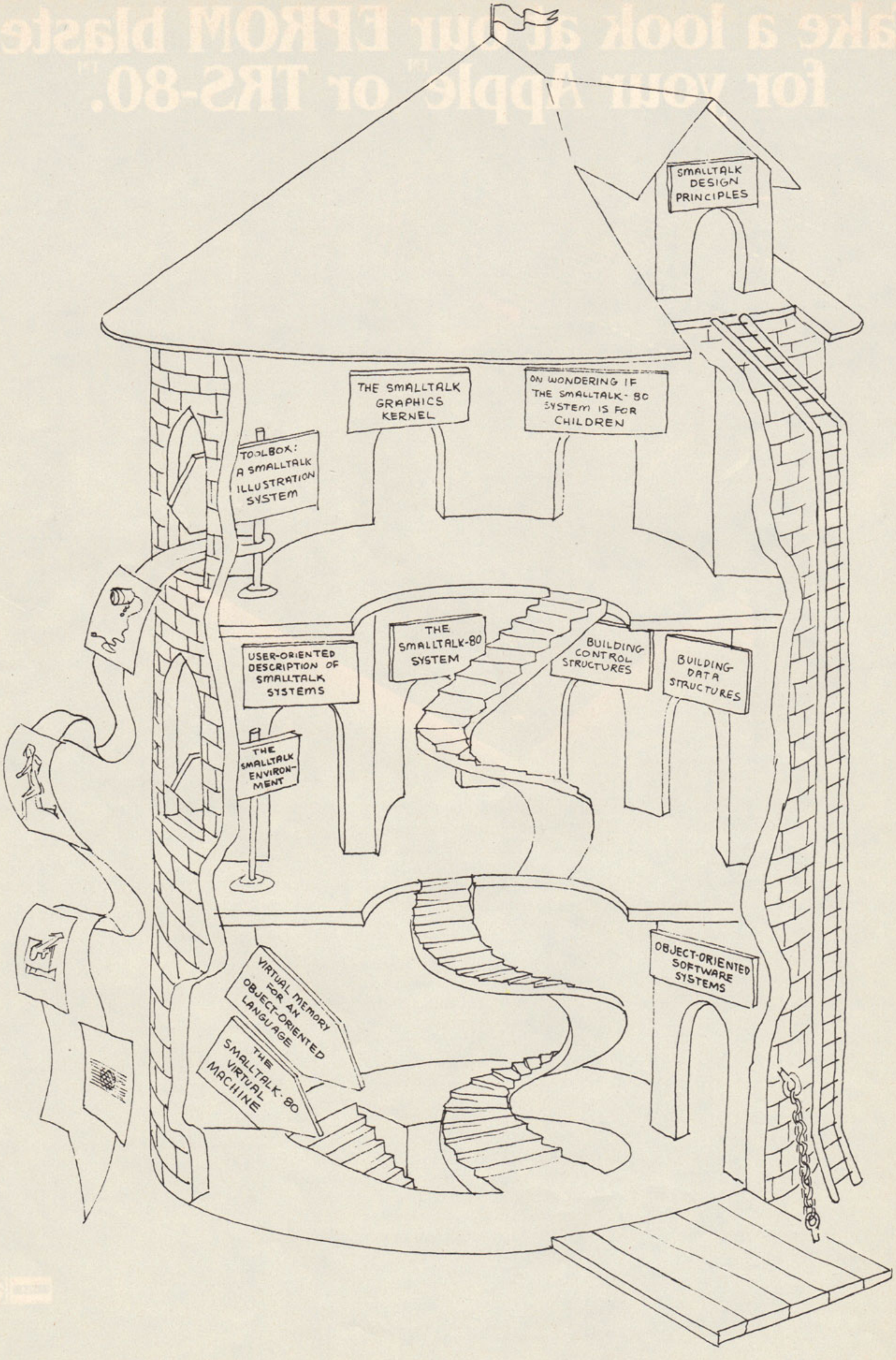

SMALLTALK DESIGN PRINCIPLES
THE SMALLTALK GRAPHICS KERNEL
ON WONDERING IF THE SMALLTALK-80 SYSTEM IS FOR CHILDREN
TOOLBOX: A SMALLTALK ILLUSTRATION SYSTEM
THE SMALLTALK-80 SYSTEM
BUILDING CONTROL STRUCTURES
BUILDING DATA STRUCTURES
USER-ORIENTED DESCRIPTION OF SMALLTALK SYSTEMS
THE SMALLTALK ENVIRON-MENT
OBJECT-ORIENTED SOFTWARE SYSTEMS
VIRTUAL MEMORY FOR AN OBJECT-ORIENTED LANGUAGE
THE SMALLTALK-80 VIRTUAL MACHINE

Figure 5

Text continued from page 22:

others, knowledge of the Smalltalk-80 system and its design philosophy is a prerequisite to understanding many of them. The map in figure 5 is presented to help the reader find a course through this hitherto uncharted ivory tower.

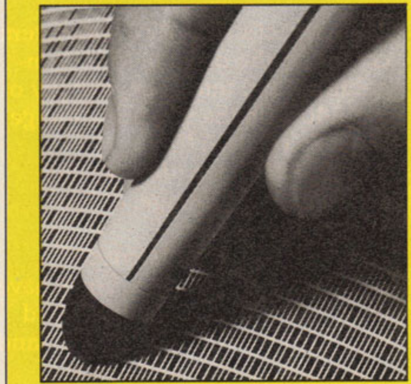
You can begin at the drawbridge by reading Dave Robson's introduction to object-oriented programming (page 74) and then proceed by reading the description of the Smalltalk-80 language (page 36). The two examples of programming in Smalltalk-80 are likely next steps: one, by Jim Althoff, tells you how to build data structures (page 230); the other, by Peter Deutsch, describes how to build control structures (page 322). Or, you can follow a hallway to the user interface window and read Larry Tesler's description of the Smalltalk programming environment (page 90). Trygve Reenskaug offers further perspectives on providing a programming interface to a Smalltalk system (page 147).

At any time, you can take the side stairs to read Dan Ingalls' presentation of the design principles behind Smalltalk (page 286). Those readers who are interested in implementation details can head for the cellar and read Glenn Krasner's article on the Smalltalk virtual machine (page 300), or Ted Kaehler's article on a Smalltalk virtual memory (page 378).

The walls of the tower are covered with visual images that will please any graphics enthusiast. Many were created by the ToolBox painting component of Smalltalk, as described in Bill Bowman and Bob Flegal's article (page 369). Greater detail about the Smalltalk graphics kernel is provided by Dan Ingalls (page 168).

Ivory towers are often associated with educational enterprises. So it is not surprising that field studies of the various versions of Smalltalk have been carried out mostly in educational settings; elementary, junior, and senior high school students as well as university students have helped us test our ideas. Joan Ross and I provide some of the history in an article exploring whether the Smalltalk-80 system is for children (page 348).

Many people have helped to build our ivory tower, to surround it with protective clouds, and then to blow some of the clouds away. All the people, past and present, of the Xerox Palo Alto Research Center contributed a brick or two. George Pake, vice president of Corporate Research, assembled the bricklayers. We especially herald the person who is responsible for laying the foundation, Alan Kay, and current members of LRG not named as article scribes: Peggy Asprey, Alan Borning, Laura Gould, Bruce Horn, Neil Jacobstein, Kim McCall, Diana Merry, Steve Putz, and Steve Weyer. Special thanks to Bert Sutherland who did the "preflight check."■



The Smalltalk-80 System

The Xerox Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

The Smalltalk-80 system represents the current state of the object-oriented point of view as it has been reduced to practice by the Xerox Learning Research Group. The Smalltalk-80 system is composed of objects that interact only by sending and receiving messages. The programmer implements a system by describing messages to be sent and describing what happens when messages are received.

The Smalltalk-80 system is the latest in a series of programming environments that have applied the object-oriented point of view more and more uniformly to the design and production of software systems. The fundamental ideas of objects, messages, and classes came from SIMULA. (See reference 1.) SIMULA allows users to create object-oriented systems, but uses the standard data/procedure-oriented ALGOL language to provide numbers, booleans, basic data structures, and control structures. The Flex system, the Smalltalk-72, Smalltalk-74, and Smalltalk-76 (see references 5, 2, and 4, respectively) systems extended the object-oriented point of view to an increasing number of the elements of a programming environment. For example, in Smalltalk-72, arithmetic, list structures, and control structures were represented as objects and messages, but classes were not. In Smalltalk-74, class descriptions as objects were introduced. The Smalltalk-76 system added the capability to express relationships between classes, and extended the object-oriented point of view to the programmer's interface.

This article presents the central semantic features and most of the syntactic features of the Smalltalk-80 system. It was prepared by Dave Robson and Adele Goldberg as scribes for the group effort of designing and implementing the system. Two forthcoming books (see reference 3) provide the full specification of the Smalltalk-80 system; in particular, the books describe the implementation of the interpreter and storage manager, and the graphical user interface.

Sending Messages—Expressions

Messages are described by *expressions*, which are sequences of characters that conform to the syntax of the Smalltalk-80 programming language. A message-sending expression describes the *receiver*, *selector*, and *arguments* of the message. When an expression is *evaluated*, the message it describes is transmitted to its receiver. Here are several examples of expressions describing a message to an object. (Note: color has been added to help identify the receivers, selectors, and arguments in the following examples.)

1. frame center
2. origin + offset
3. frame moveTo: newLocation
4. list at: index put: element

Key: 🟨 Receiver □ Selector ▨ Argument

Each expression begins with a description of the receiver of the message. The receivers in these examples are described by *variable names:* frame, origin, frame, and list, respectively. Generally, at least one space must separate the parts of an expression.

Messages without arguments are called *unary messages*. A unary message consists of a single identifier called a unary selector. The first example is a unary message whose selector is center.

A *binary message* has a single argument and a selector that is one of a set of special single or double characters called *binary selectors*. For example, the common arithmetic symbols (+, −, *, and /) are binary selectors; some comparison operations are represented as double characters (eg: == for equivalence, ~ = for not equal). The second example is a binary message whose argument is offset.

A *keyword message* has one or more arguments and a selector that is made up of a series of *keywords*, one preceding each argument. A keyword is an identifier with a trailing colon. The third example is a single-argument keyword message whose selector is moveTo: and whose argument is newLocation. The fourth example is a two-argument keyword message whose selector is made up of the keywords at: and put: and whose arguments are index and element. To talk about the selector of a multiple-argument keyword message, the keywords are concatenated. So, the selector of the fourth example is at:put:.

The message receivers and arguments in the examples are described by variable names. In addition, they can also be described with *literals*. The two most common kinds of literals are integers and strings. An *integer literal* is a sequence of digits that may be preceded by a minus sign (eg: 0, 1, 156, −3, or 13772). A *string literal* is a sequence of characters between single quotes (eg: 'hi', 'John', or 'the Smalltalk-80 system'). A binary message with an integer literal as its receiver is

```
45 + count
```

A keyword message with a string literal as its argument is

```
printer display: 'Monthly Payroll'
```

When a message is sent, it invokes a method determined by the class of the receiver. The invoked method will always return a result (an object). The result of a message can be used as a receiver or argument for another message. An example of a unary message describing the receiver of another unary message is

```
window frame center
```

Unary messages are parsed left to right. The first message in this example is the unary selector frame sent to the object named window. The unary message center is then sent to the result of the expression window frame (ie: the object returned from window's response to frame).

Binary messages are also parsed left to right. An example of a binary message describing the receiver of another binary message is

```
index + offset * 2
```

The result of sending the binary message + offset to the object named index is the receiver for the binary message *2. All binary selectors have the same precedence; only the order in which they are written matters. Parentheses can be used to change the order of evaluation. A message within parentheses is sent before any messages outside the parentheses. If the previous example were written

```
index + (offset * 2)
```

the result of the binary message * 2 to offset would be used as the argument of a binary message with receiver index and selector + .

Unary messages take precedence over binary messages. If unary messages and binary messages appear together, the unary messages will be sent first. In the example

```
frame center + window offset − index
```

the result of the unary message center to frame is the receiver of the binary message whose selector is + and whose argument is the result of the unary message offset to window. The result of the + message is, in turn, the receiver of the binary message − index. Parentheses can be used to explicitly show the order of evaluation, eg: ((frame center) + (window offset)) − index. Parentheses can also be used to alter the order of evaluation. In the example

```
(center + offset) x
```

the binary message + offset would be sent before the unary message x.

Whenever keywords appear in an unparenthesized message, they compose a single selector. The example

```
window showText: 'Title' inFont: helvetica
  indented: 15
```

is a single message whose selector is showText:inFont:indented:. Because of this concatenation, there is no left-to-right parsing rule for keyword messages. If a keyword message is to be used as a receiver or argument of another keyword message, it must be parenthesized. The expression

```
frame scale: (factor max: 5)
```

describes two keyword messages. The result of the expression factor max: 5 is the argument for the scale: message to frame.

Binary messages take precedence over keyword messages. When unary, binary, and keyword messages appear in the same expression without parentheses, the unary messages are sent first, the binary messages next, and the keyword messages last. The example

```
bigFrame height: smallFrame height * 2
```

is evaluated as if it were parenthesized as follows:

```
bigFrame height: ((smallFrame height) * 2)
```

A *cascaded message expression* describes a sequence of messages to be sent to the same object. A simple message expression is a description of the receiver (ie: a variable name, literal, or expression) followed by a message (ie: a unary selector, a binary selector and argument, or a set of keywords and arguments). A cascaded message expres-

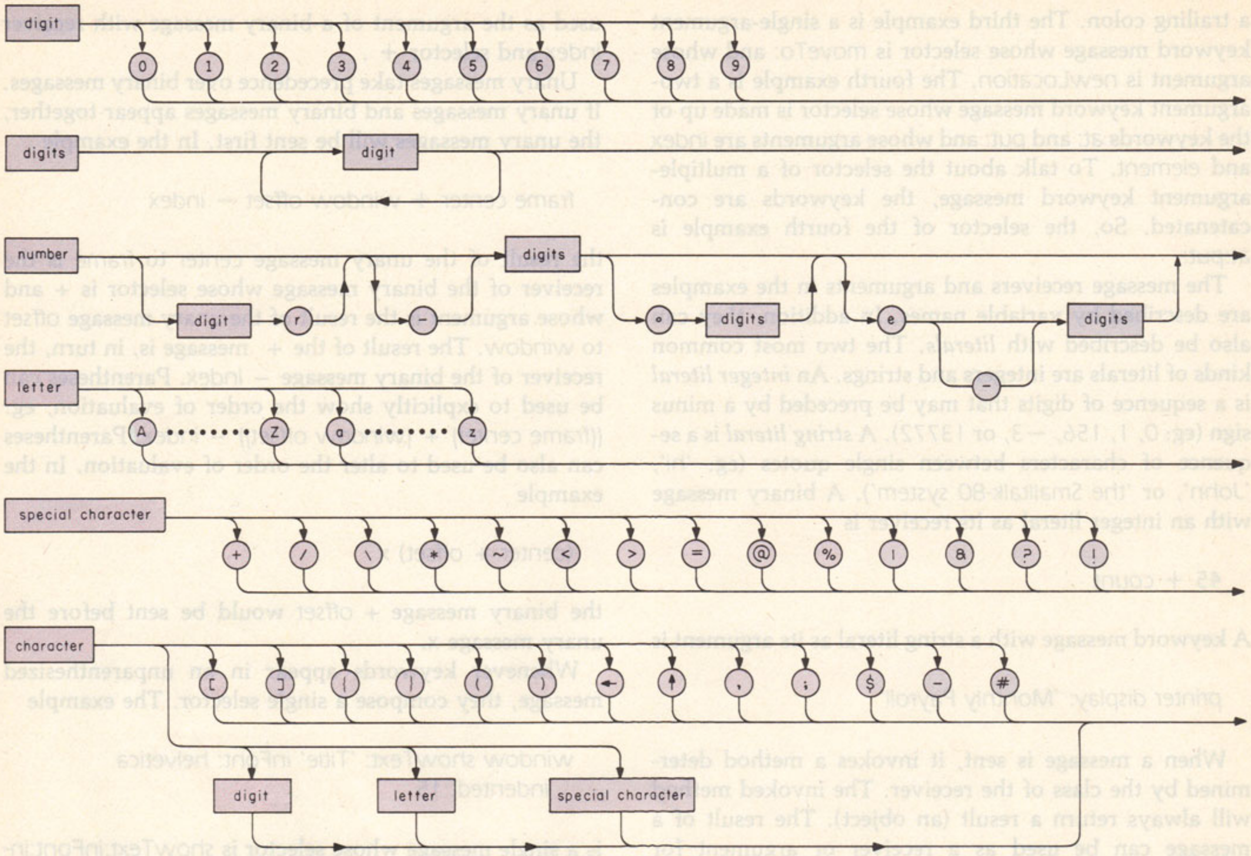


Figure 1: *Syntax diagrams for the Smalltalk-80 language.*

sion is a single description of a receiver followed by several messages separated by semicolons. For example, in the expression

```
printer newLine; print: reportTitle; space;
   print: Date today.
```

four messages are sent to the object named printer. The selectors of the four messages are newLine, print:, space, and print:. In the expression

```
window frame center: pointer location;
   width: border + contents; clear
```

three messages are sent to the object returned from the frame message to window. The selectors of the three messages are center:, width:, and clear. Without cascading, this would have been three expressions

```
window frame center: pointer location.
window frame width: border + contents.
window frame clear
```

Assigning Variables

The value of a variable can be used as the receiver or argument of a message by including its name in an expression. The value of a variable can be changed with an *assignment expression*. An assignment expression consists of a variable name followed by a left arrow (←) followed by the description of an object. When an assignment expression is evaluated, the variable named to the left of the arrow assumes the value of the object described to the right of the arrow. The new value can be described by a variable name, a literal, or a message-sending expression. Examples of assignments are

```
center ← origin
index ← 0
index ← index + 1
index ← index + 1 max: limit
```

In the last example, the message + 1 is sent to the value of the variable index, the message max: limit is sent to the result of the + message, and the result of the max: limit message becomes the new value of the variable index.

A number of variables can be assigned in the same expression by including several variable names with left arrows. The expression

```
start ← index ← 0
```

makes the value of both start and index be 0.

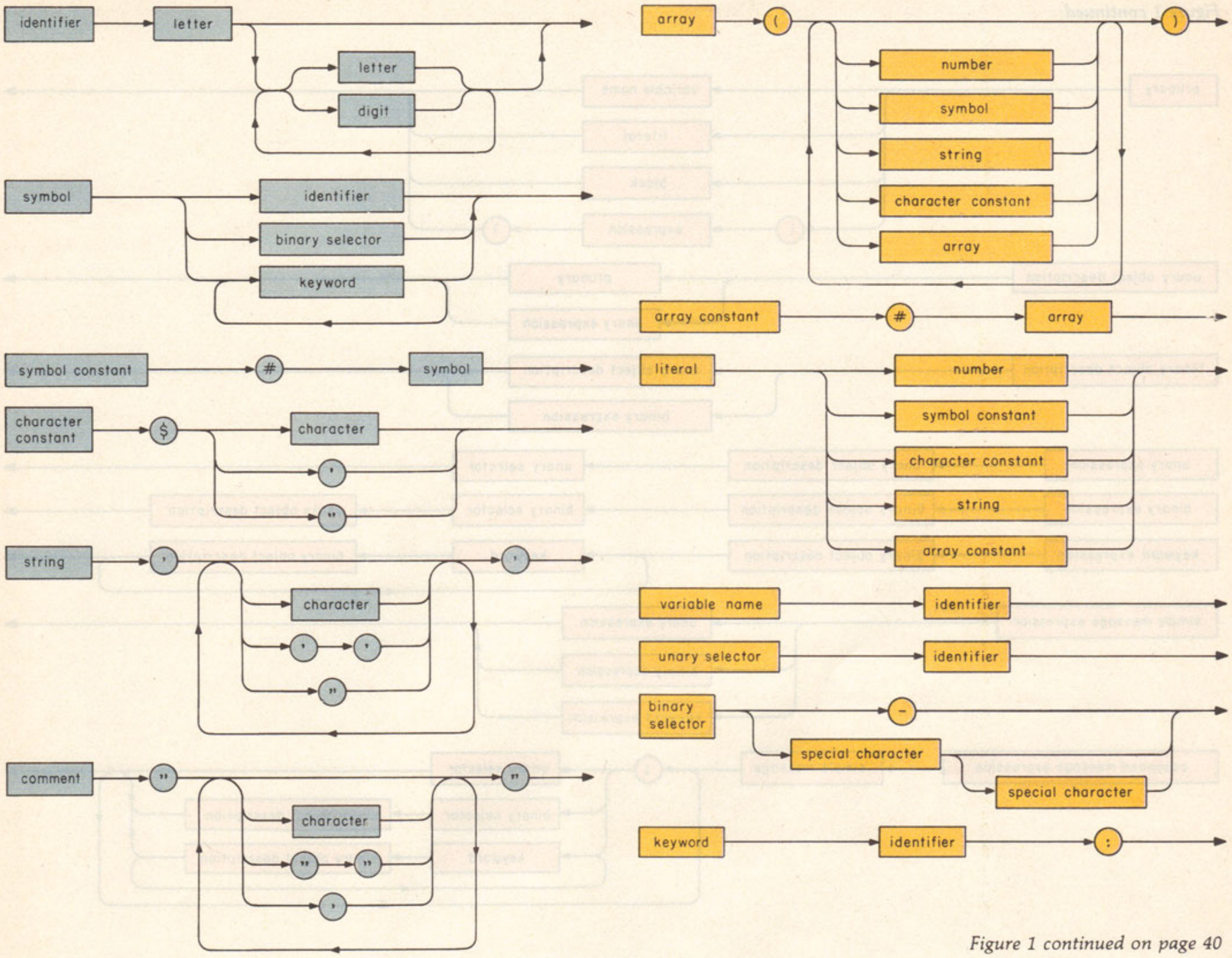


Figure 1 continued on page 40

The syntax table in figure 1 is a diagram for parsing well-formed Smalltalk-80 expressions. This table does not specify how *spaces* are treated. Spaces must not appear between digits and characters that make up a single token, nor within the specification of a number. Spaces must appear

- between a sequence of identifiers used as variables or unary selectors
- between the elements of an array in an array constant
- on either side of a keyword in a keyword expression

Spaces may optionally be included between any other elements in an expression. A carriage return or tab has the same syntactic function as a space.

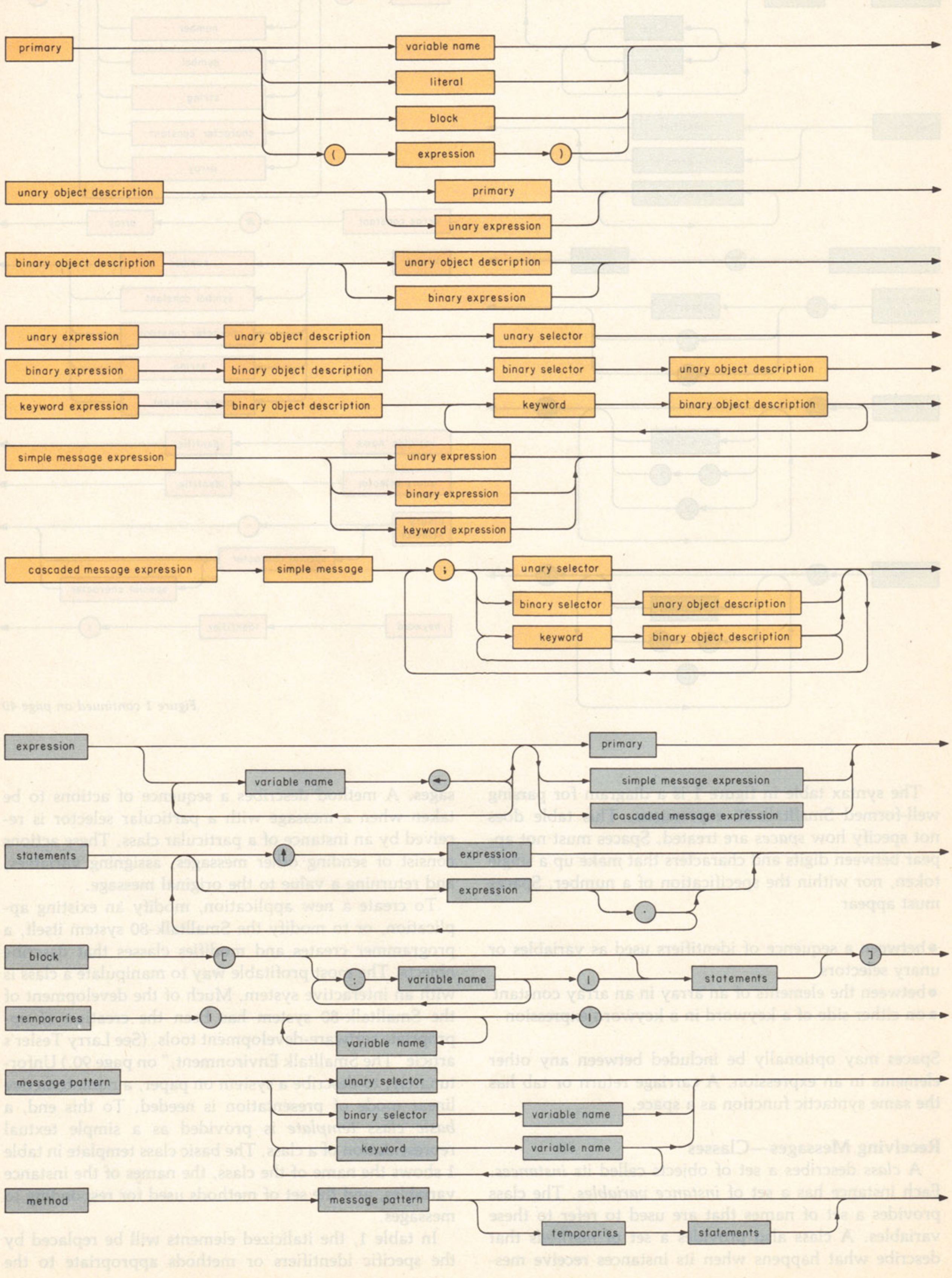
Receiving Messages—Classes

A *class* describes a set of objects called its *instances*. Each instance has a set of *instance variables*. The class provides a set of names that are used to refer to these variables. A class also provides a set of *methods* that describe what happens when its instances receive messages. A method describes a sequence of actions to be taken when a message with a particular selector is received by an instance of a particular class. These actions consist of sending other messages, assigning variables, and returning a value to the original message.

To create a new application, modify an existing application, or to modify the Smalltalk-80 system itself, a programmer creates and modifies classes that describe objects. The most profitable way to manipulate a class is with an interactive system. Much of the development of the Smalltalk-80 system has been the creation of appropriate software-development tools. (See Larry Tesler's article "The Smalltalk Environment," on page 90.) Unfortunately, to describe a system on paper, a noninteractive linear mode of presentation is needed. To this end, a *basic class template* is provided as a simple textual representation of a class. The basic class template in table 1 shows the name of the class, the names of the instance variables, and the set of methods used for responding to messages.

In table 1, the italicized elements will be replaced by the specific identifiers or methods appropriate to the

Figure 1 continued:



class name	*identifier*
instance variable names	*identifier identifier identifier*
methods	
method	
method	
method	

Table 1: *The basic class template.*

class. Names of classes begin with an uppercase letter, and names of variables begin with a lowercase letter. As an example, figure 2 shows the basic template form of a class named Point whose instances represent points in a two-dimensional coordinate system. Each instance has an instance variable named x that represents its horizontal coordinate and an instance variable named y that represents its vertical coordinate. Each instance can respond to messages that initialize its two instance variables, request the value of either variable, and perform simple arithmetic. The details of methods (in particular, the use of '|', '.' and '↑') are the subject of our next discussion.

Methods

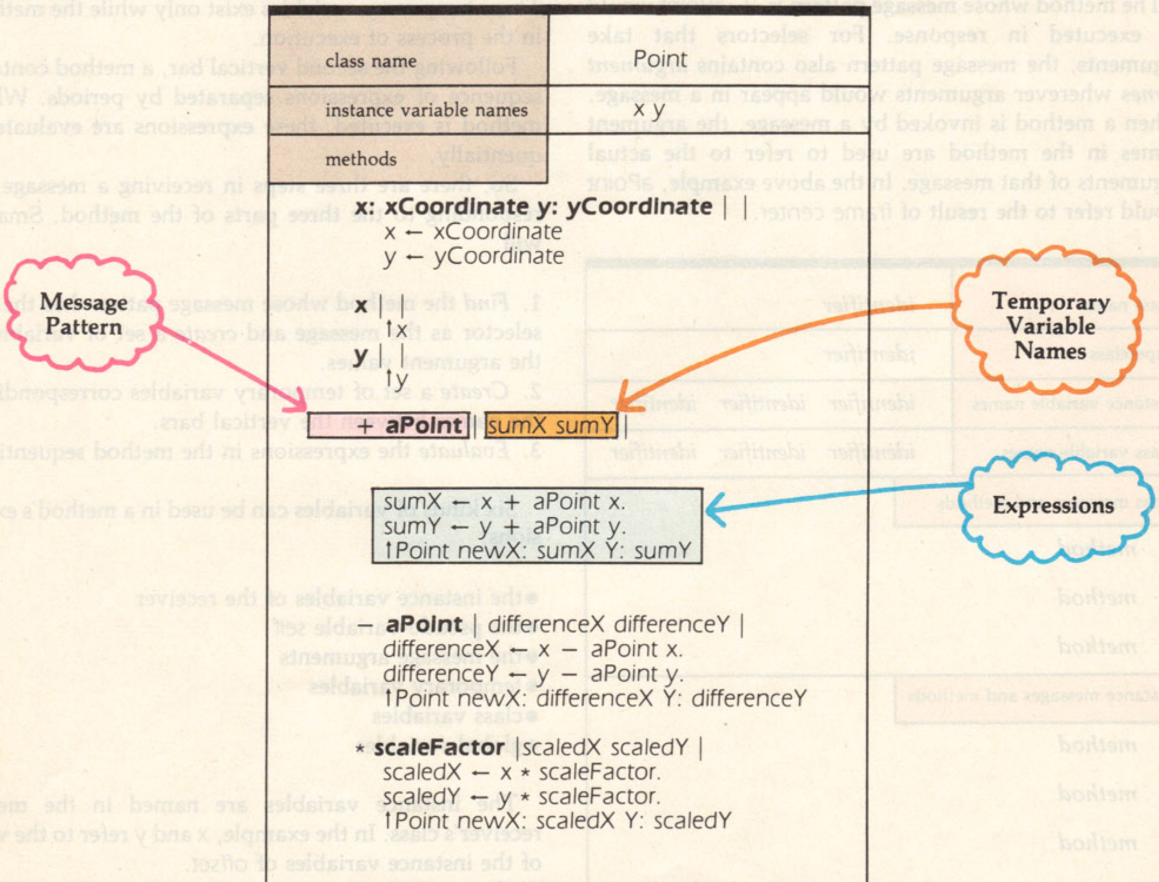
A method has three parts:

- a *message pattern*
- some *temporary variable names*
- some expressions

The three parts of a method are separated by vertical bars (|). The message pattern consists of a selector and names for the arguments. The expressions are separated by periods (.) and the last one may be preceded by an up arrow (↑). In the method for selector + in figure 2, the message pattern is + aPoint, the temporary variable names are sumX and sumY, and there are three expressions, the last one preceded by an ↑.

Line breaks have no significance in methods; formatting is used only for purposes of aesthetics. The vertical bars and periods are delimiters of significance.

As stated earlier, each message pattern contains a selector. When a message is received by an instance, the method whose message pattern contains the same selector will be executed. For example, suppose that offset were an



```
class name                Point
instance variable names   x y
methods

x: xCoordinate y: yCoordinate | |
    x ← xCoordinate
    y ← yCoordinate

x | |
    ↑x
y | |
    ↑y

+ aPoint | sumX sumY |

    sumX ← x + aPoint x.
    sumY ← y + aPoint y.
    ↑Point newX: sumX Y: sumY

– aPoint | differenceX differenceY |
    differenceX ← x – aPoint x.
    differenceY ← y – aPoint y.
    ↑Point newX: differenceX Y: differenceY

* scaleFactor |scaledX scaledY |
    scaledX ← x * scaleFactor.
    scaledY ← y * scaleFactor.
    ↑Point newX: scaledX Y: scaledY
```

Figure 2: *Illustrated class template for the class* Point.

class name	DepositRecord
superclass	Object
instance variable names	date amount
methods	

```
of: depositAmount on: depositDate | |

    date ← depositDate.
    amount ← depositAmount

amount | |
    ↑ amount

balanceChange | |
    ↑ amount
```

Table 2: *Class template for class* DepositRecord.

instance of Point in the expression

offset + frame center

The method whose message pattern is + aPoint would be executed in response. For selectors that take arguments, the message pattern also contains *argument names* wherever arguments would appear in a message. When a method is invoked by a message, the argument names in the method are used to refer to the actual arguments of that message. In the above example, aPoint would refer to the result of frame center.

class name	*identifier*
superclass	*identifier*
instance variable names	*identifier identifier identifier*
class variable names	*identifier identifier identifier*
class messages and methods	
method	
method	
method	
instance messages and methods	
method	
method	
method	

Table 4: *The full class template.*

class name	CheckRecord
superclass	DepositRecord
instance variable names	number
methods	

```
number: checkNumber for: checkAmount on:
  checkDate | |
    number ← checkNumber.
    date ← checkDate.
    amount ← checkAmount

of: anAmount on: aDate | |
    self error:
      'Check records are initialized with
      number:for:on:'

balanceChange | | ↑ 0 − amount
```

Table 3: *Class template for class* CheckRecord.

Following the message pattern, a method can contain some *temporary variable names* between vertical bars. When a method is executed, a set of variables is created that can be accessed by the temporary variable names. These temporary variables exist only while the method is in the process of execution.

Following the second vertical bar, a method contains a sequence of expressions separated by periods. When a method is executed, these expressions are evaluated sequentially.

So, there are three steps in receiving a message, corresponding to the three parts of the method. Smalltalk will

1. *Find* the method whose message pattern has the same selector as the message and *create* a set of variables for the argument values.
2. *Create* a set of temporary variables corresponding to the names between the vertical bars.
3. *Evaluate* the expressions in the method sequentially.

Six kinds of variables can be used in a method's expressions:

- the instance variables of the receiver
- the pseudo-variable self
- the message arguments
- temporary variables
- class variables
- global variables

The instance variables are named in the message receiver's class. In the example, x and y refer to the values of the instance variables of offset.

There is an important *pseudo-variable* available in every method, which is named self. self refers to the receiver of the message that invoked the method. It is called a pseudo-variable because its value can be accessed like a variable, but its value cannot be changed using an assignment expression. In the example, self refers to the same object as offset during the execution of the method associated with +.

Arguments and temporary variables are similar, in that the names for both are declared in the method itself and they both exist only during the method's execution. However, unlike arguments, temporary variables are not automatically initialized. The values of temporary variables can be changed with an assignment expression.

Class variables are shared by all instances and the class itself. Names for the class variables are shown in the full class template in an entry called "class variable names" (see table 4). Although they are variables and their values can be changed, they are typically treated as constants, initialized when the class is created, and then simply used by the instances. For example, if the class of floating-point numbers wanted to provide trigonometric functions, it might want to define a variable called pi to be used in any of its methods.

Global variables are shared by all objects. A global dictionary, called Smalltalk, holds the names and values of these variables. The classes in the system, for example, are the values of global variables whose names are the class names. With the exception of variables used to reference system resources, few global variables exist in the Smalltalk-80 system. Programming style that depends on user-defined globals is generally discouraged.

If the last expression in a method is preceded by an ↑, the message that invoked the method takes on the value of this expression. If an ↑ does not precede the last expression, the value of the message is simply the receiver of the message. For example, the x:y: message to a Point (see figure 2) behaves as if it had been written

```
x: xCoordinate y: yCoordinate | |
    x ← xCoordinate.
    y ← yCoordinate.
    ↑ self
```

Methods can contain comments anywhere. A *comment* is a sequence of characters delimited by double quotes. Two consecutive double quotes are used to embed a double quote within a comment. The methods in class Point were purposely written in a verbose style to provide examples. The messages for + could have been written

```
+ aPoint | |
    ↑ Point newX: x + aPoint x Y: y + aPoint y
```

The basic class template presents only the most important attributes of a class. The complete description of a class is provided by the *full class template*, described in the next section.

Inheritance

The basic template allows a class to be described independently of other classes. It ignores inheritance among classes. The full class template, however, takes inheritance into account. (See table 4.) With it, a class can be described as a modification of another class called its *superclass*. All classes that modify a particular class are called its *subclasses*. A subclass inherits the instance variable names and methods of its superclass. A subclass can also add instance variable names and methods to those it inherits. The instance variable names added by the subclass must differ from the instance variable names of the superclass. The subclass can *override* a method in the superclass by adding a message with the same selector. Instances of the subclass will execute the method found in the subclass rather than the method inherited from the superclass.

To assemble the complete description of a class, it is necessary to look at its superclass, its superclass's superclass, and so on, until a class with no superclass is encountered. There is only one such class in the system (ie: without a superclass), and its class name is Object. All classes ultimately inherit methods from Object. Object has no instance variables. The set of classes linked through the superclass relation is called a *superclass chain*. The full class template has an entry called "superclass" that specifies the initial link on the class's superclass chain.

As an example, we might describe a class, DepositRecord, whose instances are records of bank account deposits. Each instance has two instance variables representing the date and amount of the deposit. The class template is shown in table 2.

class name	CheckRecord
superclass	DepositRecord
instance variable names	number
class messages and methods	

```
number: checkNumber for: checkAmount on:
    checkDate | |
    ↑ self new number: checkNumber
              for: checkAmount
              on: checkDate
```

instance messages and methods

```
number: checkNumber for: checkAmount on:
    checkDate | |
    super of: checkAmount on: checkDate.
    number ← checkNumber

of: anAmount on: aDate | |
    self error: 'Check records are initialized with
        number:for:on:'

balanceChange | | ↑ 0 − amount
```

Table 5: *Full class template for class* CheckRecord.

A class, CheckRecord, whose instances are records of checks written on an account is a subclass of DepositRecord; this new class adds an instance variable that represents the check number. The class template is shown in table 3.

An instance of CheckRecord has three instance variables. It inherits the amount message, adds the number:for:on: message, and overrides the balanceChange and of:on: messages. The of:on: method contains a single expression in which the message error: 'Check records are initialized with number:for:on:' is sent to the pseudo-variable self. The method for error: is found in the superclass of DepositRecord, which is the class Object; the response is to stop execution and to display the string literal argument to the user.

An additional pseudo-variable available in a method's expressions is super. It allows a subclass to access the methods in its superclass that have been overridden in the subclass description. The use of super as the receiver of a message has the same effect as the use of self, except that the search for the appropriate message starts in the superclass, not the class, of the receiver.

class name	Point
superclass	Object
instance variable names	x y
class variable names	pi
class messages and methods	

```
instance creation
newX: xValue Y: yValue | |
   ↑ self new x: xValue
              y: yValue
newRadius: radius Angle: angle | |
   ↑ self new x: radius * angle sin
              y: radius * angle cos

class initialization
setPI | | pi ← 3.14159
```

instance messages and methods	

```
accessing
x: xCoordinate y: yCoordinate | |
   x ← xCoordinate.
   y ← yCoordinate
x | | ↑x
y | | ↑y
radius | | ↑((x * x) + (y * y)) squareRoot
angle | | ↑(x/y) arctan

arithmetic
+ aPoint | | ↑Point newX: x + aPoint x
                        Y: y + aPoint y
− aPoint | | ↑Point newX: x − aPoint x
                        Y: y − aPoint y
* scaleFactor | | ↑Point newX: x * scaleFactor
                             Y: y * scaleFactor
circleArea | r |
   r ← self radius.
   ↑ pi * r * r
```

Table 6: *Full class template for class* Point.

For example, the method associated with number:for:on in CheckRecord might have been defined as

```
number: checkNumber for: checkAmount on:
checkDate | |
   super of: checkAmount on: checkDate.
   number ← checkNumber
```

Metaclasses

Since a class is an object, there is a different class that describes it. A class that describes a class is called a *metaclass*. Thus, a class has its own instance variables that represent the description of its instances; it responds to messages that provide for the initialization and modification of this description. In particular, a class responds to a message that creates a new instance. The unary message new creates a new instance whose instance variables are uninitialized. The object nil indicates an uninitialized value.

The classes in the system might all be instances of the same class. However, each class typically uses a slightly different message protocol to create initialized instances. For example, the last expression in the method associated with + in class Point (see figure 2) was

```
Point newX: sumX Y: sumY
```

newX:Y: is a message to Point, asking it to create a new instance with sumX and sumY as the values of the new instance's instance variables. The newX:Y: message would not mean anything to another class, such as DepositRecord or CheckRecord. So, these three classes can't be instances of the same class. All classes have a lot in common, so their classes are all subclasses of the same class. This class is named Class. The subclasses of Class are called *metaclasses*.

The newX:Y: message in Point's metaclass might be implemented as

```
newX: xValue Y: yValue |
   ↑ self new x: xValue y: yValue
```

The new message was inherited by Point's metaclass from Class. One reason for having metaclasses is to have a special set of methods for each class, primarily messages for initializing class variables and new instances. These methods are displayed in the full class-template form shown in table 4; they are distinguished from the methods for messages to the instances of the class. The two categories are "class messages and methods" and "instance messages and methods," respectively. Methods in the category "class messages and methods" are associated with the metaclass; those in "instance messages and methods" are associated with the class.

If there are no class variables for the class, the "class variable name" entry is omitted. So, CheckRecord might be described as shown in table 5.

It is often desirable to create subcategories within the categories "class messages and methods" and "instance messages and methods." Moreover, the order in which the categories or subcategories are listed is of no significance. (The notion of categories is simply a pretty printing" technique; it has no semantic significance.)

Returning to the example of class Point, if the instance methods of class Point include subcategories *accessing* and *arithmetic*, the template for Point might appear as shown in table 6.

When the class Point is defined, the expression

```
Point setPi
```

should be evaluated in order to set the value of the single class variable.

A Point might be created and given a name by evaluating the expression

```
testPoint ← Point newX: 420 Y: 26
```

The new Point, testPoint, can then be sent the message circleArea:

```
testPoint circleArea
```

or used in a more complex expression:

```
(testPoint * 2) circleArea
```

Primitive Routines

The response to some messages in the system may be performed by a *primitive routine* (written in the implementation language of the machine) rather than by evaluating the expressions in a method. The methods for these messages indicate the presence of such a primitive routine by including <primitive> before the first expression in the method. A major use of primitive methods is to interact with the machine's input/output devices.

An example of a primitive method is the new message to classes, which returns a new instance of the receiver.

```
new | | <primitive>
```

This particular primitive routine always produces a result. If there are situations in which a primitive routine cannot produce a result, the method will also contain some expressions. If the primitive routine is successful in responding to the message, it will return a value and the expressions in the method will not be evaluated. If the primitive routine encounters difficulty, the expressions will be evaluated as though the primitive routine had not been specified.

Another example of a message with a primitive response is a message with the selector + sent to a SmallInteger

```
+ aNumber | | <primitive>
    self error: 'SmallInteger addition has failed'
```

One reason this primitive might fail to produce a result is that the argument is not a SmallInteger. In the example, this would produce an error report. In the actual Smalltalk-80 system, an attempt is made to check and see if the argument were another kind of number for which a result could be produced.

Indexed Instance Variables

An object's instance variables are usually given names by its class. The names are used in methods of the class to refer to the values of the instance variables. Some objects also have a set of instance variables that have no names and can only be accessed by messages. The instance variables are referred to by an integral *index*. Indexable objects are used to implement the classes in the system that represent collections of other objects, such as arrays and strings.

The messages to access indexed instance variables have

class name	Array
superclass	IndexedCollection
indexable instance variables	
class messages and methods	
instance creation **with: anElement** | | ↑(self new: 1) at: 1 put: anElement **with: firstElement with: secondElement** | anArray | anArray ← self new: 2. anArray at: 1 put: firstElement. anArray at: 2 put: secondElement. ↑anArray	
instance messages and methods	
accessing **at: anInteger** | | < primitive > self error: 'index out of range' **at: anInteger put: anElement** | | < primitive > self error: 'index out of range' *funny stuff* **embed** | | ↑Array with: self	

Table 7: *Full class template for class* Array.

selectors at: and at:put:. For example

```
list at: 1
```

returns the first indexed instance variable of list. The example

```
list at: 4 put: element
```

stores element as the value of the fourth indexed instance variable of list. The at: and at:put: messages invoke primitive routines to load or store the value of the indicated variable. The legal indices run from one to the number of indexable variables in the instance. The at: and at:put: messages are defined in class Object and, therefore, can be understood by all objects; however, only certain classes will create instances with indexable instance variables. These classes will have an additional line in the class template indicating that the instances contain *indexable instance variables*. As an example, we show a part of the template for class Array in table 7.

Each instance of a class that allows indexable instance variables may have a different number of them; such instances are created using the new: message to a class, whose argument tells the number of indexable variables. The number of indexable instance variables an instance has can be found by sending it the message size. A class whose instances have indexable instance variables can also have named instance variables. All instances of any class will have the same number of named instance variables.

Control Structures and Blocks

The two control structures in the Smalltalk-80 system described so far are

- the sequential execution of expressions in a method
- the sending of messages that invoke other methods that eventually return values

All other control structures are based on objects called *blocks*. Like a method, a block is a sequence of expressions, the last of which can be preceded by an up arrow (↑). The expressions are delimited by periods; they may be preceded by one or more identifiers with leading colons. These identifiers are the *block arguments*. Block arguments are separated from expressions by a vertical bar.

Whenever square brackets are encountered in a method, a block is created. Evaluation of the expressions inside the square brackets is deferred until the block is sent the message value or a message whose selector is a concatenation of one or more occurrences of the keyword value:. Control structures are implemented as messages with receivers or arguments that are blocks. The methods for carrying out these control-structure messages involve sending the blocks patterns of value messages.

In the Smalltalk-80 system, there are two types of primitive control messages: conditional selection of blocks, ifTrue:ifFalse:, and conditional iteration of blocks, whileTrue: and whileFalse:.

The representation of conditions in the Smalltalk-80 system uses distinguished boolean objects named false and true. The first type of primitive control message provides for conditional selection of a block to be executed. This is similar to the IF . . . THEN . . . ELSE of ALGOL-like languages. The expression

```
queue isEmpty ifTrue: [index ← 0]
              ifFalse: [index ← queue next]
```

evaluates the expressions in the first block if the receiver is true and evaluates the expressions in the second block if the receiver is false. Two other forms of conditional selection provide only one alternative

```
queue isEmpty ifTrue: [index ← 0].
queue isEmpty ifFalse: [index ← queue next].
```

When ifTrue: is sent to false, it returns immediately without executing the block. When ifFalse: is sent to true, the block is not executed.

The second type of primitive control message repeatedly evaluates the expressions in a block as long as some condition holds. This is similar to the WHILE and UNTIL statements in ALGOL-like languages. This type of control message is a message to a block; the receiver, the block, evaluates the expressions it contains and determines whether or not to continue on the basis of the value of the last expression. The first form of this control message has selector whileTrue:. The method for whileTrue: repeatedly executes the argument block as long as the receiver's value is true. For example,

```
[index < = limit] whileTrue: [self process: list at: index.
                              index ← index + 1]
```

The binary message < = is understood by objects representing magnitudes. The value returned is the result of comparing whether the receiver is less than or equal to (< =) the argument.

The second conditional iteration message has selector whileFalse:. The method for whileFalse: repeatedly executes the argument block as long as the receiver's value is false. For example,

```
[queue isEmpty] whileFalse: [self process: queue next]
```

The messages whileTrue and whileFalse to a block provide a shorthand notation for messages of the form whileTrue: aBlock and whileFalse: aBlock, if the argument aBlock is an empty block.

Block arguments allow one or more of the variables inside the block to be given new values each time the block is executed. Instead of sending the block the message value, messages with selectors value: or value:value:, and so on, are sent to the block. The arguments of the value: messages are assigned to the block arguments (in order) before the block expressions are evaluated.

As an example, classes with indexed instance variables could implement a message with selector do: that takes a block as an argument and executes it once for every indexed variable. The block has a single block argument; the value of the appropriate indexed variable is passed to it for each execution. An example of the use of such a message is

```
list do: [ :element | self process: element]
```

The message might be implemented as

```
do: aBlock | index |
    index ← 1.
    [index < = self size] whileTrue:
          [aBlock value: (self at: index).
            index ← index + 1]
```

Similar control messages can be implemented for any class. As an example, a simple repetition could be provided by a timesRepeat: aBlock message to instances of class Integer

```
timesRepeat: aBlock | index |
    index ← 1.
    [index < = self] whileTrue:
          [aBlock value.
            index ← index + 1]
```

Examples of implementing other control messages are given in L Peter Deutsch's article "Building Control Structures in the Smalltalk-80 System," on page 322.

The Smalltalk-80 System: Basic Classes

The Smalltalk-80 *language* provides a uniform syntax for retrieving objects, sending messages, and defining classes. The Smalltalk-80 *system* is a complete programming environment that includes many actual classes and instances. In support of the uniform syntax, this system includes class descriptions for Object, Class, Message, CompiledMethod, and Context, whose subclasses are BlockContext and MethodContext. Multiple independent processes are provided by classes ProcessorScheduler, Process, and Semaphore. The special object nil is the only instance of class UndefinedObject. These classes comprise the *kernel* Smalltalk-80 system.

The system also includes class descriptions to support basic data structures; these are numerical and collection classes. The class Number specifies the protocol appropriate for all numerical objects. Its subclasses provide specific representations of numbers. The subclasses are Float, Fraction, and Integer. For a variety of reasons, there are both SmallIntegers and LargeIntegers; of these, there are LargePositiveIntegers and LargeNegativeIntegers.

Class Collection specifies protocol appropriate to objects representing collections of objects. These include Bag, Set, OrderedCollection, LinkedList, MappedCollection, SortedCollection, and IndexedCollection. The latter provides protocol for objects with indexable instance variables. It has subclasses String and Array. Elements of a string are instances of class Character; bytes are stored in instances of ByteArray. A subclass of String is Symbol; a subclass of Set is Dictionary (a set of Associations).

Interval is a subclass of Collection with elements representing an arithmetic progression. Intervals can be created by sending the message to: or to:by: to Integer. So, the expressions 1 to: 5 by: 1 and 1 to: 5 each create a new Interval representing 1, 2, 3, 4, 5. As a Collection, Interval responds to the enumeration message do:. For example, in

```
(1 to: 5) do: [:index | anArray at: index put: index * 2]
```

the block argument index takes on successive values 1, 2, 3, 4, 5.

For programmer convenience, an Integer also responds to the messages to:do: and to:by:do:, allowing the parentheses in interval enumeration expressions to be omitted.

The ability to stream over indexed or ordered collections is provided by a hierarchy based on class Stream, including ReadStream, WriteStream, and ReadAndWriteStream. A file system, local or remote, is then implementable as a subclass of these kinds of Streams.

Since instances of the system classes described above are used in the implementation of all applications, an understanding of their message protocol is as necessary to understanding an implementation as an understanding of the language syntax. These system classes are fully described in the forthcoming Smalltalk books.

In addition to the basic data-structure classes, the Smalltalk-80 system includes class descriptions to support interactive graphics (forms and images and image editors, text and text editors), networking, standard files, and hard-copy printing. A complete Smalltalk-80 system contains about sixty class definitions, not including a variety of windows or views, menus, scrollbars, and the metaclasses. Many of these are discussed in companion articles in this issue. (See Daniel H H Ingalls's "The Design Principles Behind Smalltalk," page 286, and Larry Tesler's "The Smalltalk Environment," page 90.)

The important thing to note is that each of these class descriptions is implemented in the Smalltalk-80 language itself. Each can be examined and modified by the programmer. Some of the class descriptions contain methods that reference primitive methods; only these methods are implemented in the machine language of the implementation machine. It is a fundamental part of the philosophy of the system design that the programmer have such complete access. In this way, system designers, such as members of the Xerox Learning Research Group, are able to build the next Smalltalk in the complete context of Smalltalk itself. ■

References

1. Birtwistle, Graham; Ole-Johan Dahl; Bjorn Myhrhaug; and Kristen Nygaard. *Simula Begin*. Philadelphia: Auerbach, 1973.
2. Goldberg, Adele and Alan Kay, editors. *Smalltalk-72 Instructional Manual*. Xerox PARC technical report, March 1976 (out of print).
3. Goldberg, Adele; David Robson; and Daniel H H Ingalls. *Smalltalk-80: The Language and Its Implementation* and *Smalltalk-80: The Interactive Programming Environment*, 1981 (books forthcoming).
4. Ingalls, Daniel H H. "The Smalltalk-76 Programming System: Design and Implementation." In *Proceedings of the Principles of Programming Languages Symposium*, January 1978.
5. Kay, Alan. *The Reactive Engine*. Ph.D. Thesis, University of Utah, September, 1969 (University Microfilms).

Glossary

Editor's Note: *This glossary provides concise definitions for many of the keywords and concepts related to Smalltalk-80. These definitions will be most useful if you first read the introductory Smalltalk articles. . . .* **GW**

General Terminology

object — *a package of information and descriptions of its manipulation*

message — *a specification of one of an object's manipulations*

method — *a procedure-like entity; the description of a sequence of actions to be taken when a message is received by an object*

class — *a description of one or more similar objects*

instance — *an object described by a particular class*

method dictionary — *a set of associations between message selectors and methods; included in each class description*

metaclass — *a class whose (single) instance is itself a class*

subclass — *a class that is created by sharing the description of another class, often modifying some aspects of that description*

Syntax Terminology

message receiver — *the object to be manipulated, according to a message*

message sender — *the object requesting a manipulation*

message selector — *a symbolic name that describes a desired manipulation of an object*

message argument — *one of the objects specified in a message that provides information needed so that a message receiver can be manipulated appropriately*

unary message — *a message without arguments*

binary message — *a message with a single argument and a selector that is one of a set of special single or double characters*

keyword message — *a message that has one or more arguments and a selector made up of a series of identifiers with trailing colons, one preceding each argument*

block — *a literal method; an object representing a sequence of actions to be taken at a later time, upon receiving an "evaluation" message (such as one with selector* value *or* value:*)*

Semantics

instance variable — *a variable that is information used to distinguish an instance from other instances of the same class*

class variable — *a variable shared by all instances of a class and the class itself*

named variable — *an instance variable that is given a name in the class of the instance; the name is used in methods of the class*

indexed variable — *an instance variable with no name, accessed by message only; referred to by an integer (an index)*

global or pool variable — *a variable shared by instances of several classes; a system example is* Smalltalk, *a dictionary that includes references to all the defined classes*

temporary variable — *a variable that exists only while the method in which it is declared is in the process of execution*

pseudo-variable — *a variable available in every method without special declaration, but whose value cannot be changed using an assignment. System examples are* self, super *and* thisContext.

nil — *a special object, the only instance of class* UndefinedObject

Implementation Terminology

field — *the memory space in which the value of an object's variable is stored*

bytecode — *a machine instruction for the virtual machine*

object pointer — *a reference to an object*

reference count — *of an object, is the number of objects that point to it (ie: that contain its object pointer)*

Circle 336 on inquiry card. →

Object-Oriented Software Systems

David Robson
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

This article describes a general class of tools for manipulating information called object-oriented software systems. It defines a series of terms, including software system *and* object-oriented. *The description is greatly influenced by a series of object-oriented programming environments developed in the last ten years by the Learning Research Group of Xerox's Palo Alto Research Center, the latest being the Smalltalk-80 system. The article describes object-oriented software systems in general, instead of the Smalltalk-80 system in particular, in order to focus attention on the fundamental property that sets the Smalltalk-80 system apart from most other programming environments. The words "object-oriented" mean different things to different people. Although the definition given in this article may exclude systems that should rightfully be called object-oriented, it is a useful abstraction of the idea behind many software systems.*

Many people who have no idea how a computer works find the idea of object-oriented systems quite natural. In contrast, many people who have experience with computers initially think there is something strange about object-oriented systems. (I don't mean to imply that computer-naive users can create complex systems in an object-oriented environment more easily than experienced programmers can. Creating complex systems involves many techniques more familiar to the programmer than the novice, regardless of whether or not an object-oriented environment is used. But the basic idea about how to create a software system in an object-oriented fashion comes more naturally to those without a preconception about the nature of software systems.) I had had some programming experience when I first encountered an object-oriented system and the idea certainly seemed strange to me. I am assuming that most of you also have some experience with software systems and their creation. So instead of introducing the object-oriented point of view as if it were completely natural, I'll try to explain what makes it seem strange compared to the point of view found in other programming systems.

Software Systems

A software system is a tool for manipulating information. For the purposes of this article, I'm using a very broad definition of information.

> **Information:** *A representation or description of something.*

There are many types of information that describe different things in different ways. One of the great insights in computer science was the fact that information can (among other things) describe the manipulation of information. This type of information is called *software*.

> **Software:** *Information describing the manipulation of information.*

Software has the interesting recursive property of describing how to manipulate things like itself. Software is used to describe a particular type of information-manipulation tool called a *software system*.

> **Software system:** *An information-manipulation tool in which the manipulation is described by software.*

A distinction is made in information-manipulation tools between *hardware* systems and *software* systems. A hardware system is a physical device like a typewriter, pen, copier, or television set. The type of manipulation performed by a hardware system is built in and can only be changed by physical modification. The type of manipulation performed by a software system is not built in—it is determined by information, which can be manipulated.

The virtue of software systems is that the mechanism developed for manipulating information can be used to manipulate the mechanism itself. Software systems that actually manipulate other software systems are called *programming environments*.

> **Programming environment:** *A software system that manipulates software systems. An environment for the design, production, and use of software systems.*

Thus, a programming environment is also recursive: it *is* what it manipulates. The fact that software systems can be manipulated is both good news and bad news. Since a text editor is a software system, it is not "cast in concrete" and you can change it to conform to your style of interacting with text more closely than it does now (using a programming environment). However, you also may reduce it to the proverbial "pile of bits" (not a text editor at all).

Data/Procedure-Oriented Software

The traditional view of software systems is that they are composed of a collection of *data* that represents some information and a set of *procedures* that manipulates the data.

> **Data:** *The information manipulated by software.*
>
> **Procedure:** *A unit of software.*

Things happen in the system by invoking a procedure and giving it some data to manipulate.

As an example of a software system, consider a system for managing windows that occupy rectangular areas on a display screen. The windows contain text and have titles. They can be moved around the screen, sometimes overlapping each other. (The details of this system are not important. Its main purpose is to point out the differences between the structure of a data/procedure system and an object-oriented system.)

A window-management system implemented as a data/procedure system would include data representing the location, size, text contents, and title of each window on the screen. It would also include procedures that move a window, create a window, tell whether a window overlaps another window, replace the text or title of a window, and perform other manipulations of windows on a display. To move a window, a programmer would call the procedure that moves windows and pass to it the data representing the window and its new location.

A problem with the data/procedure point of view is that data and procedures are treated as if they were independent when, in fact, they are not. All procedures make assumptions about the form of the data they manipulate. The procedure to move a window should be presented with data representing a window to be moved and its new location. If the procedure were presented with data representing the text contents of a window, the system would behave strangely.

In a properly functioning system, the appropriate choice of procedure and data is always made. However, in an improperly functioning system (eg: one in the process of being developed or encountering an untested situation), the data being manipulated by a procedure may be of an entirely different form from that expected. Even in a properly functioning system, the choice of the appropriate procedure and data must always be made by the programmer.

These two problems have been addressed in the context of the data/procedure point of view by adding several features to programming systems. Data typing has been added to languages to let the programmer know that the appropriate choice of data has been made for a particular procedure. In a typed system, the programmer is notified when a procedure call is written using the wrong type of data. Variant records allow the system to choose the appropriate procedure and data in some situations.

Object-Oriented Software

Instead of two types of entity that represent information and its manipulation independently, an object-oriented system has a single type of entity, the *object*, that represents both. Like pieces of data, objects can be manipulated. However, like procedures, objects describe manipulation as well. Information is manipulated by sending a *message* to the object representing the information.

> **Object:** *A package of information* and *descriptions of its manipulation.*
>
> **Message:** *A specification of one of an object's manipulations.*

When an object receives a message, it determines how to manipulate itself. The object to be manipulated is called the *receiver* of the message. A message includes a symbolic name that describes the type of manipulation desired. This name is called the message *selector*. The crucial feature of messages is that the selector is only a *name* for the desired manipulation; it describes *what* the programmer wants to happen, not *how* it should happen. The message receiver contains the description of how the actual manipulation should be performed. The programmer of an object-oriented system sends a message to invoke a manipulation, instead of calling a procedure. A message names the manipulation; a procedure describes the details of the manipulation.

Of course, procedures have names as well, and their names are used in procedure calls. However, there is only one procedure for a name, so a procedure name specifies the exact procedure to be called and exactly what should happen. A message, however, may be interpreted in different ways by different receivers. So, a message does not determine exactly what will happen; the receiver of the message does.

If the earlier example of the window-management system were implemented as an object-oriented system, it would contain a set of objects representing windows. Each object would describe a window on the screen. Each object would also describe the manipulations of the window it represents—for example, how to move it, how to determine whether it overlaps another window, or how to display it. Each of these manipulations would correspond to a selector of a message. The selectors could include move, overlap, display, delete, width, or height. (In this article, an alternate typeface is used for words that refer to specific elements in example systems.)

In addition to a selector, a message may contain other objects that take part in the manipulation. These are called the message *arguments*. For example, to move a

window, the programmer might send the object representing the window a message with the selector move. The message would also contain an argument representing the new location. Since this is an object-oriented system, the selector and argument are objects: the selector representing a symbolic name and the argument representing a location or point.

The description of a single type of manipulation of an object's information (the response to a single type of message) is a procedure-like entity called a *method*. A method, like a procedure, is the description of a sequence of actions to be performed by a processor. However, unlike a procedure, a method cannot directly call another method. Instead, it must send a message. *The important thing is that methods cannot be separated from objects.* When a message is sent, the receiver determines the method to execute on the basis of the message selector. A different kind of window could be added to the system with a different representation and different methods to respond to the messages move, overlap, display, delete, width, and height. Places where messages are sent to windows do not have to be changed in order to refer to the new kind of window; whichever window receives the message will use the method appropriate to its representation.

Objects look different from the outside than they do from the inside. By the outside of an object, I mean what it looks like to other objects with which it interacts (eg: what rectangles look like to other rectangles or to windows). From the outside, you can only ask an object to do something (send it a message). By the inside of an object, I mean what it looks like to the programmer implementing its behavior. From the inside, you can tell an object how to do something (in a method). For example, a window can respond to messages having the selectors move, overlap, display, delete, width, or height.

However, nothing is known outside the window about how it responds to these messages. (It is known that a window will move when asked to, but it is not known how it accomplishes the move.)

The set of messages an object can respond to is called its *protocol*. The external view of an object is nothing more than its protocol; the internal view of an object is something like a data/procedure system. An object has a set of variables that refers to other objects. These are called its *private variables*. It also has a set of methods that describes what to do when a message is received. The values of the private variables play the role of data and the methods play the role of procedures. This distinction between data and procedures is strictly localized to the inside of the object.

Methods, like other procedures, must know about the form of the data they directly manipulate. Part of the data a method can manipulate are the values of its object's private variables. For example, we might imagine three ways that a window represents its location and size (internally). The private variables might contain:

- four numbers representing the *x* and *y* location of the center, the width, and the height
- two points representing opposite corners of the window
- a single rectangle whose location and size are the same as the window's

The method that moves a window (the response to messages with the selector move) assumes that a particular representation is used. If the representation were changed, the method would also have to be changed. Only the methods in the object whose representation changed need be changed. All other methods must manipulate the window by sending it messages.

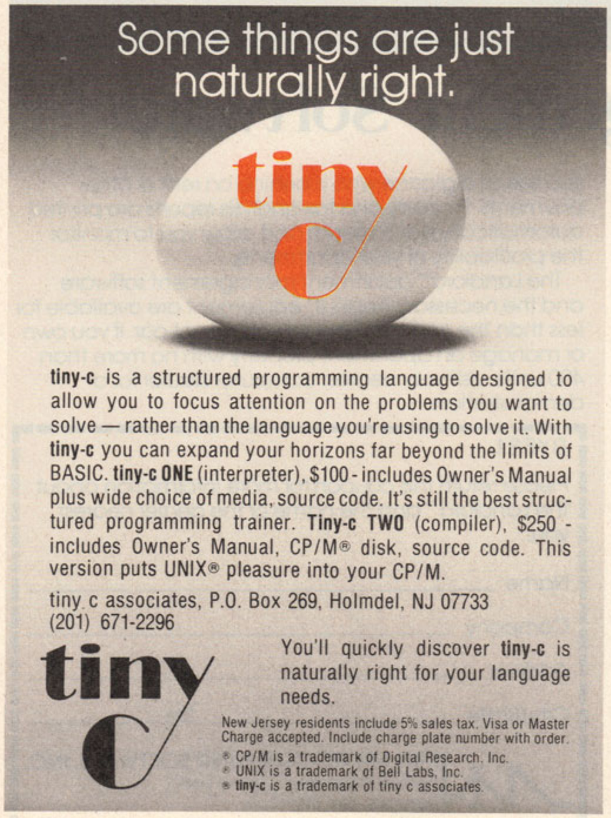
A message must be sent to an object to find out anything about it (ie: our concept of manipulation includes inquiring about information, as well as changing information). This is needed because we don't want the form of an object's inside known outside of it. The response to a message may return a value. For example, a window's response to the message width returns an object that represents its width on the display (a number). The method for determining what to return depends on the form of the window's private variables. If they are represented as the first alternative listed above (four numbers), the response would simply return the value of the appropriate private variable. If the second alternative is used (two points), the method would have to determine the width from the *x* coordinates of the two corners. If the third alternative is used (one rectangle), the width message would simply be passed on to the rectangle and the rectangle's response would become the window's response.

Classes and Instances

Most object-oriented systems make a distinction between the description of an object and the object itself. Many similar objects can be described by the same general description. The description of an object is called a *class* since the class can describe a whole set of related objects. Each object described by a class is called an *instance* of that class.

> **Class:** *A description of one or more similar objects.*
>
> **Instance:** *An object described by a particular class.*

Every object is an instance of a class. The class describes all the similarities of its instances. Each instance contains the information that distinguishes it from the other instances. This information is a subset of its private variables called *instance variables*. All instances of a class have the same *number* of instance variables. The *values* of the instance variables are different from instance to instance. An object's software (ie: the methods that describe its response to messages) is found in its class. All instances of a class use the same method to respond to a particular type of message (ie: a message with a particular selector). The difference in response by two different instances is a result of their different instance variables. The methods in a class use a set of names to refer to the set of instance variables. When a message is sent, those names in the invoked method refer to the instance variables of the message receiver. Some of an object's private



variables are shared by all other instances of its class. These variables are called *class variables* and are part of the class.

The programmer developing a new system creates the classes that describe the objects that make up the system. The programmer of the window-management system would create a class that contained methods corresponding to the message selectors move, display, delete, width, and height. This class would also include the names of the instance variables referred to in those methods. These names might be frame, text, and title, where:

> frame is a rectangle defining the area on the screen,
> text is the string of characters displayed in the window, and
> title is the string of characters representing the window's name

The classes representing rectangles and strings of characters are included in most systems, so they don't need to be defined.

In a system that is uniformly object-oriented, a class is an object itself. A class serves several purposes. In a running system, it provides the description of how objects behave in response to messages. The processor running an object-oriented system looks at the receiver's class when a message is sent to determine the appropriate method to execute. For this use of classes, it is not necessary that they be represented as objects since the processor does not interact with them through messages (preventing a nasty recursion). In a system under development, a class provides an interface for the programmer to interact with the definition of objects. For this use of classes, it is extremely useful for them to be objects, so they can be manipulated in the same way as all other descriptions. Classes also are the source of new objects in a running system. Here again, it is useful for the class to be an object, so object creation can be accomplished with a simple message. For example, the message new might be sent to a class to create a new instance.

Inheritance

Another mechanism used for implicit sharing in object-oriented systems is called *inheritance*. One object inherits the attributes of another object, changing any attributes that distinguish the two. Some object-oriented systems provide for inheritance between all objects, but most provide it only between classes. A class may be modified to create another class. In such a relationship, the first class is called the *superclass* and the second is called the *subclass*. A subclass inherits everything about its superclass. The following modifications can be made to a subclass:

- adding instance variables
- providing new methods for some of the messages understood by the superclass
- providing methods for new messages (messages not understood by the superclass)
- adding class variables

As an example, the window-management system might contain windows that have a minimum size. These would be instances of a subclass of the ordinary class of windows that added an instance variable to represent the minimum size and provided a new method for the message that changes a window's size.

Conclusion

The realization that information can describe the manipulation of information is largely responsible for the great utility of computers today. However, that discovery is also partially responsible for the failure of computers to reach the utility of some predictions made in earlier times. On the one hand, it can be seen as a unification between the manipulator and the manipulated. However, in practice, it has been seen as a distinction between software and the information it manipulates. For small systems, this distinction is harmless. But for large systems, the distinction becomes a major source of complexity. The object-oriented point of view is a way to reduce the complexity of large systems without placing additional overhead on the construction of small systems. ■

The Smalltalk Environment

Larry Tesler
Apple Computer Inc
10260 Bandley Dr
Cupertino CA 95014

As I write this article, I am wearing a T-shirt (photo 1) 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 screen is, appropriately enough, very blue.

My friend gave me the shirt to make fun of a near-fanatical campaign I have waged for several years, a campaign to eliminate modes from the 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, and 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 (reference 1).

Photo 1: *The "DON'T MODE ME IN" T-shirt.*

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 commands 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 erased from the screen and irretrievably lost from view. In general, "running a program" in most systems puts you into a mode where the facilities of other programs are unavailable to you. Dan Swinehart calls this the *dilemma of preemption* (reference 2).

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 gives 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 the 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 the *editor*, issue the *copy-from* command, and when it prompts you with "from-what-file?," simply type the name (you haven't

Circle 269 on inquiry card. ⟶

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 eliminated 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 compete 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 (photo 2). 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 that 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 diverse 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 en-

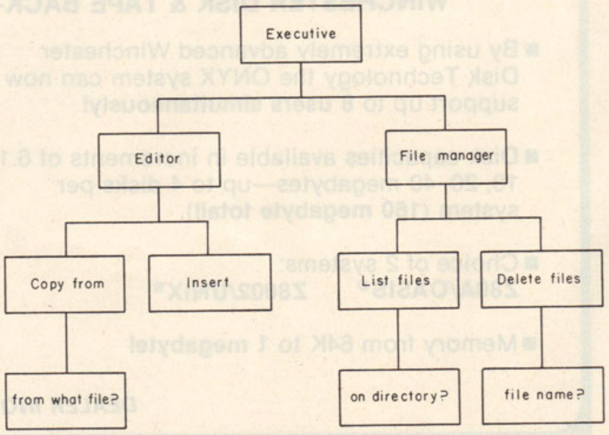


Figure 1: *A portion of a typical mode hierarchy.*

vironment. 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

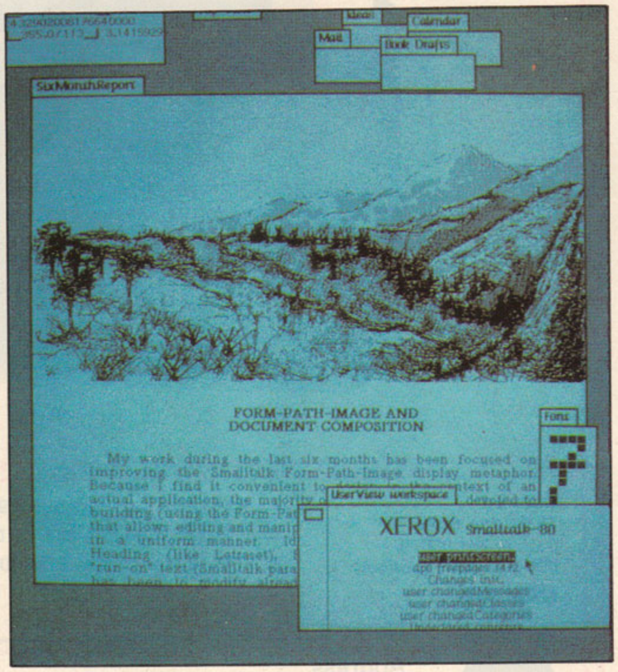
Before delving further into the *nature* of the Smalltalk environment, we should first discuss its *purpose*.

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 programs on very small computers.

All these languages have been used for numerous purposes in addition to those mentioned. 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, each language stands out above the others when used for the purpose 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 (photo 3a). 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 (photo 3b). One button is used as a *selection button*. If there are more buttons, they are normally used as *menu buttons*.

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 of 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



FORM-PATH-IMAGE AND
DOCUMENT COMPOSITION

My work during the last six months has been focused on improving the Smalltalk Form-Path-Image display metaphor. Because I find it convenient to [illegible] the context of an actual application, the majority [illegible] devoted to building (using the Form-Pat[illegible] that allows editing and manip[illegible] in a uniform manner. Id[illegible] Heading (like Letraset), [illegible] "run-on" text (Smalltalk para[illegible] has been to modify alread[illegible]

Photo 2: *A typical Smalltalk display. The various "windows" look and behave like overlapping sheets of paper.*

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 user-written 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 the keyboard.

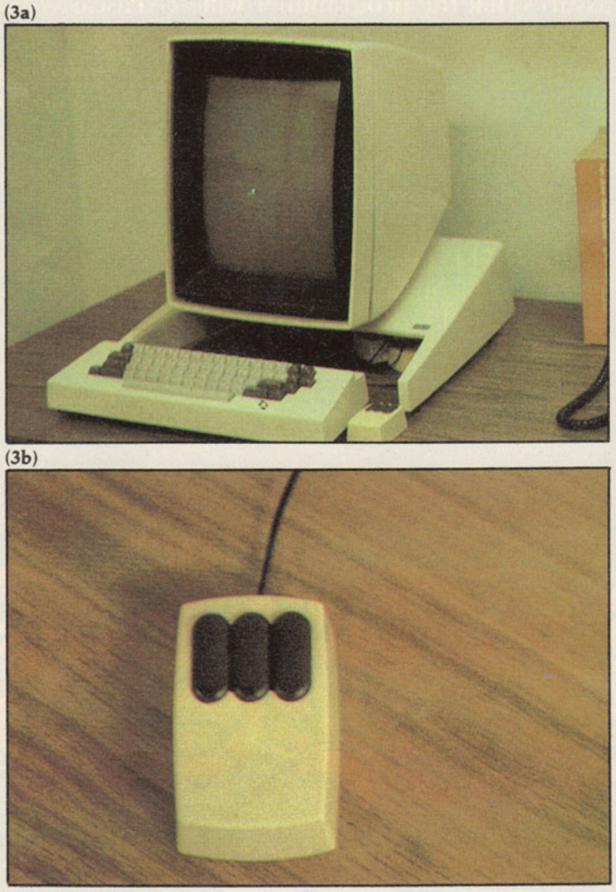

(3a)
(3b)

Photo 3: *A typical Smalltalk system (photo 3a) and a close-up of the "mouse" (photo 3b), a device that allows you to move an on-screen cursor and select certain options.*

- 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 with 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* (photo 4a). 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* (photo 4b). The pane menu includes commands appropriate to the contents of that pane. In

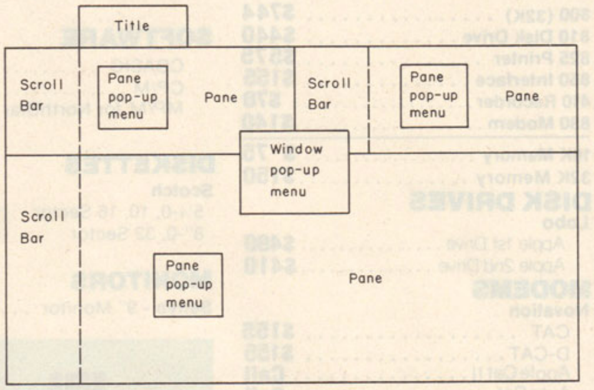
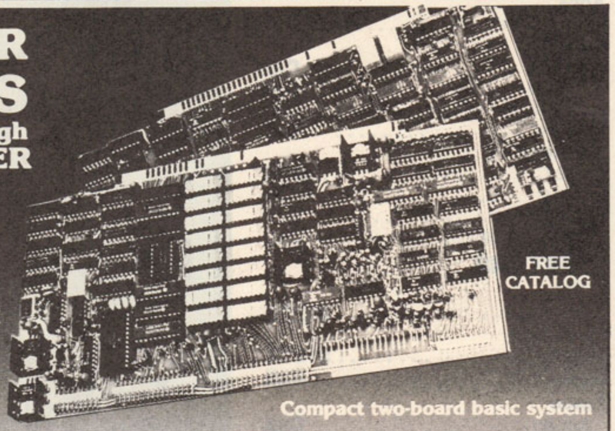
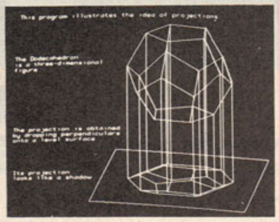
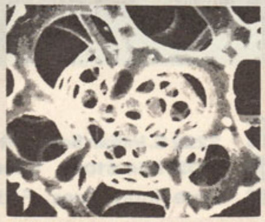


Figure 2: *Anatomy of a window.*




This program illustrates the idea of projections
The Dodecahedron is a three-dimensional figure
The projection is obtained by dropping perpendiculars onto a level surface
Its projection looks like a shadow



addition, a pane has a *scroll bar* on its left side used to scroll the contents of the pane when more information exists than fits in the frame at one time.

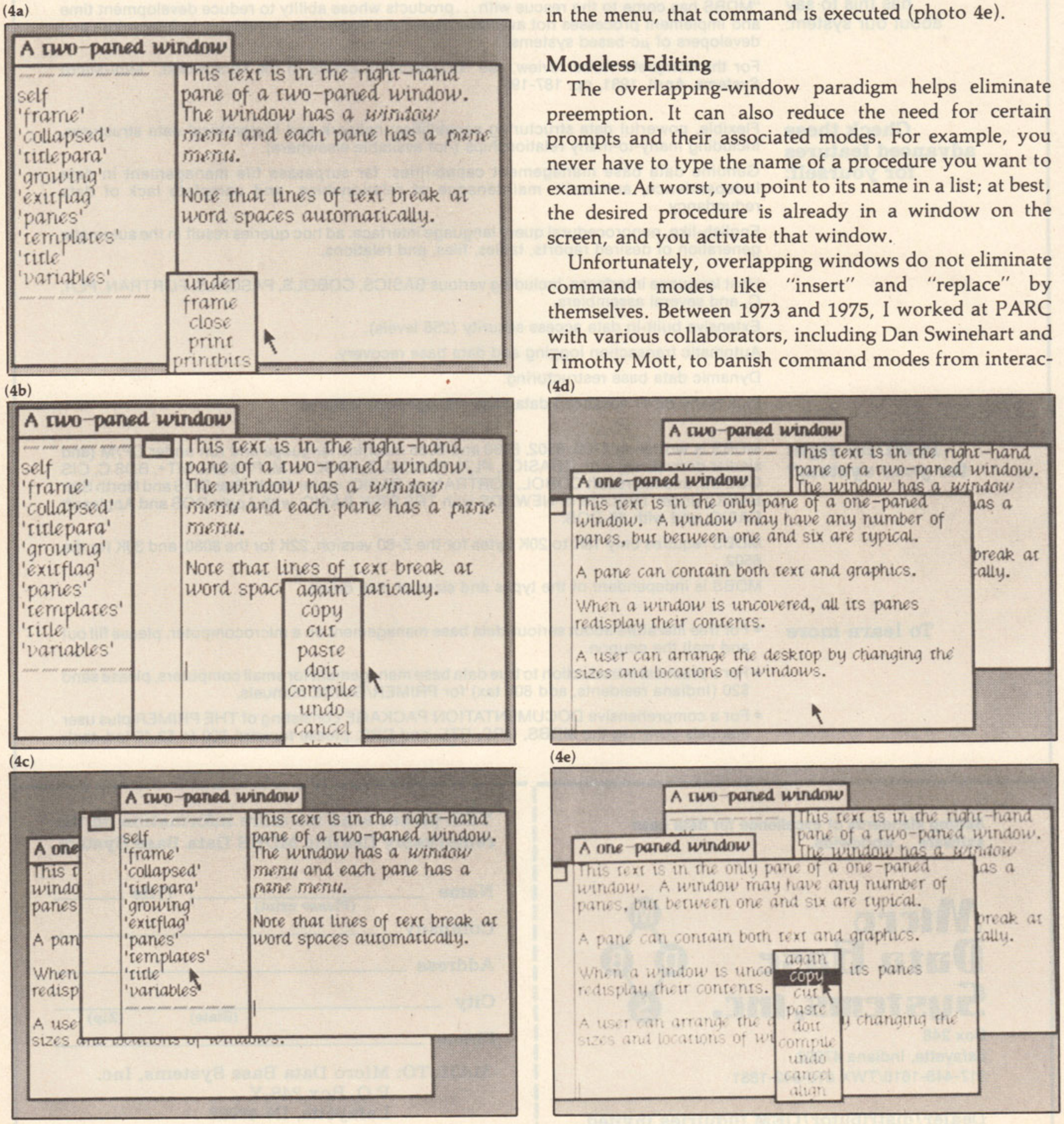
Although you can see many windows and panes at once, you can interact with only one pane at a time. That pane and its window are said to be awake or *active*. To awaken a different pane of the same window, move the cursor over the new pane (photo 4c). To awaken a different window, move the cursor over the new window and press the *selection button* on the pointing device (photo 4d). 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 versions of Smalltalk even hide the active menus until one of two *menu buttons* 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 is over a command in the menu, that command is executed (photo 4e).

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 interac-



self
'frame'
'collapsed'
'titlepara'
'growing'
'exitflag'
'panes'
'templates'
'title'
'variables'

This text is in the right-hand pane of a two-paned window. The window has a *window menu* and each pane has a *pane menu*.

Note that lines of text break at word spaces automatically.

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

When a window is uncovered, all its panes redisplay their contents.

A user can arrange the desktop by changing the sizes and locations of windows.

Photo 4: *Windows and their behavior.*

Circle 165 on inquiry card. →

tive 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.

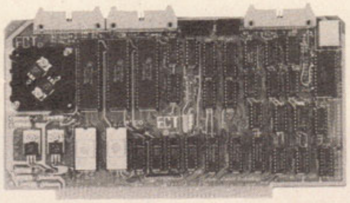
Selection precedes command:

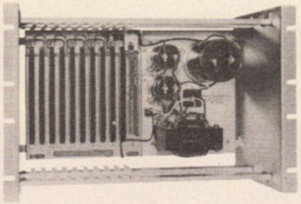
- Every command is executed immediately when you issue it. You are not asked to confirm it. You can issue an *undo* command to reverse the effects of the last issued command. Although the main purpose of "undo" is to compensate for the lack of command confirmation, it can also be used to change your 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 form later and finish filling it out.

Typing text always replaces the selected characters:

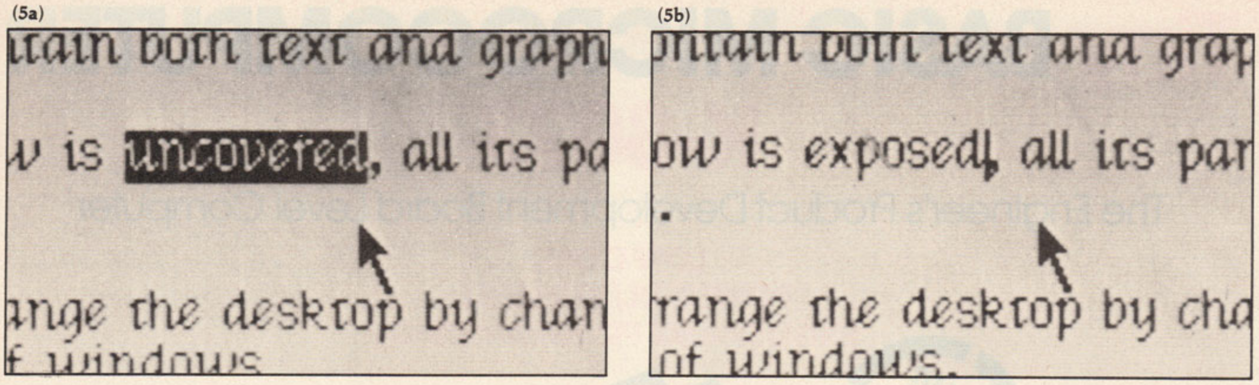
- Pressing a text key on the keyboard never issues a command. It always replaces the current selection by the typed character and automatically selects the gap following that character.
- To replace a passage of text, first select it (photo 5a) and then type the replacement. The first keystroke deletes the original text (photo 5b).
- To insert between characters, you first select the gap between those characters (photo 6a) and then type the insertion (photo 6b). 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 (photos 6c and 6d) can be used to reverse the effects of all your typing and backspacing 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.





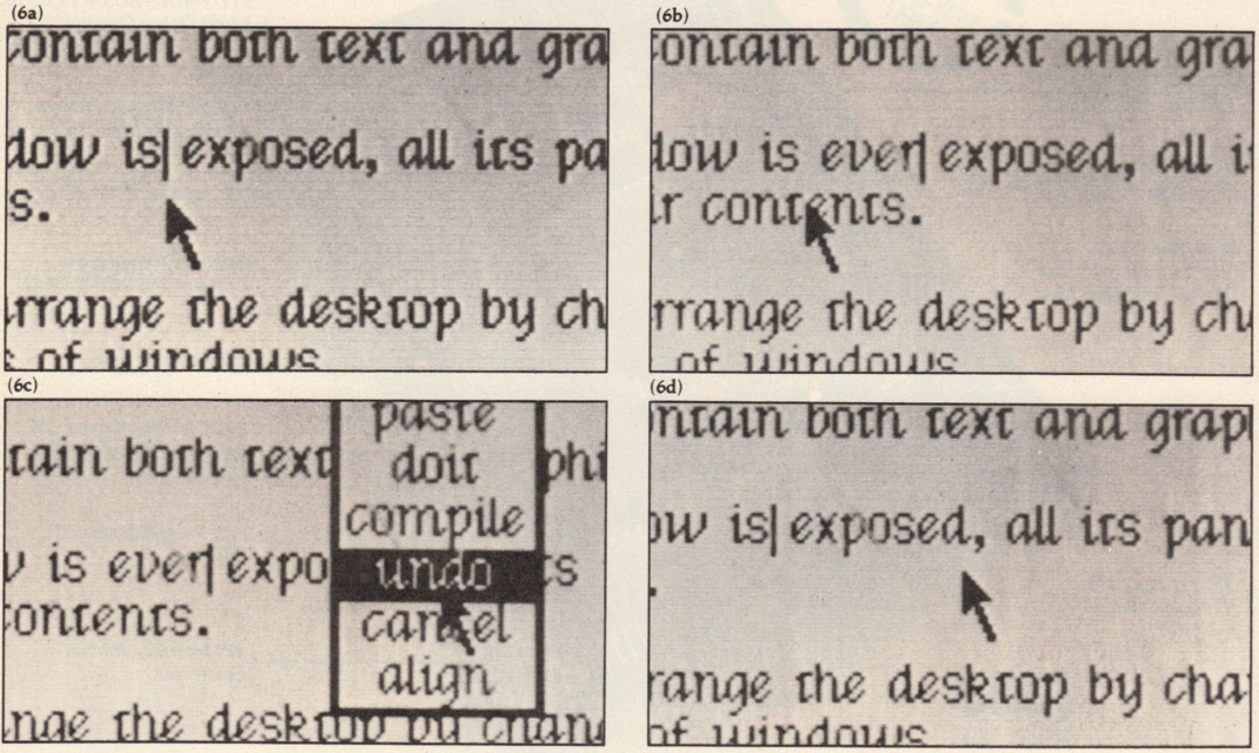
Circle 388 on inquiry card. →



tain both text and graph
v is uncovered, all its pa
nge the desktop by chan
f windows

ontain both text and grap
ow is exposed, all its par
.
range the desktop by cha
of windows.

Photo 5: *Replacing text in Smalltalk.*



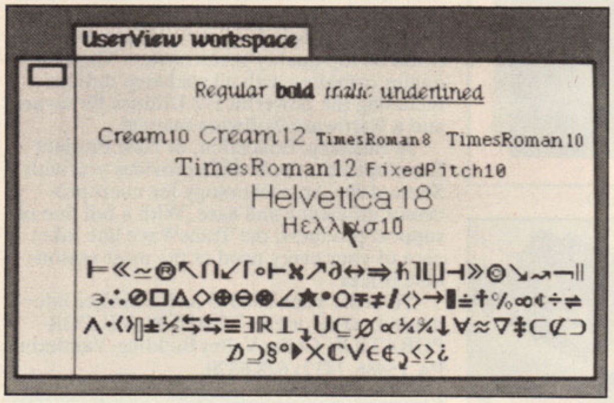
contain both text and gra
dow is| exposed, all its pa
s.
rrange the desktop by ch
of windows

contain both text and gra
dow is ever| exposed, all i
r contents.
rrange the desktop by ch
of windows

tain both text phi
v is ever| expo s
contents.
nge the desktop by chan

ntain both text and grap
ow is| exposed, all its pan
.
range the desktop by cha
of windows

Photo 6: *Inserting text in Smalltalk.*



Regular **bold** *italic* underlined

Cream10 Cream12 TimesRoman8 TimesRoman10
TimesRoman12 FixedPitch10
Helvetica18
Hελλησ10
⊢≪≃⊛↖∩↙Γ∘⊢✗↗∂↔⇒ħ⅂⊔⊣≫⊚↘⤳⊣∥
϶∴⊘□△◇⊕⊖⊗∠★∘○∓≠/‹›→▮±†%∞¢÷≠
∧·‹›▯±⅙⇆⇆≡∃ℝ⊥⊔⊆∅∝⅓⅓↓∀≈∇‡⊂⊄⊃
⊅⊃§°▸✕ℂ∨∈∈⌐≤≥¿

Photo 7: *Multiple typefaces can be used in any window.*

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**," or "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 (photo 7). Thus, as 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 to existing text to change it to boldface.

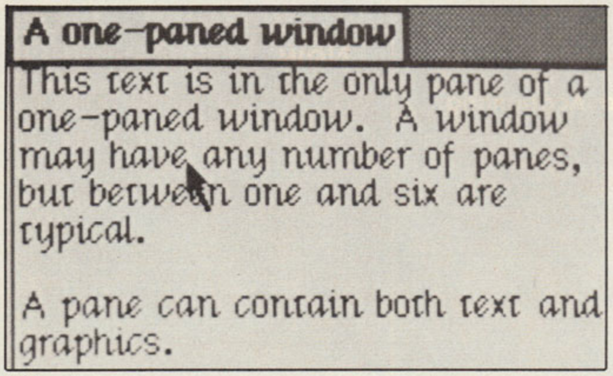
In 1976, Dan Ingalls devised a user interface for Smalltalk that incorporated most of the mode-avoidance techniques discussed earlier. Consequently, it is rare in the present Smalltalk environment to encounter a mode.

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 the passage to be selected (photo 8a). The selection button is pressed and held down 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 the cursor reaches the other end of the passage, the button is released. The selected passage is then shown in inverse video (photo 8d).

The feedback given to the user during selection is as follows. When the button is depressed, a vertical bar appears in the nearest intercharacter gap (photo 8b). (At the left end of a line of text, the bar appears to the left of the first character. At the right end of a line, the bar appears to the left of the final space character.)

(8a)

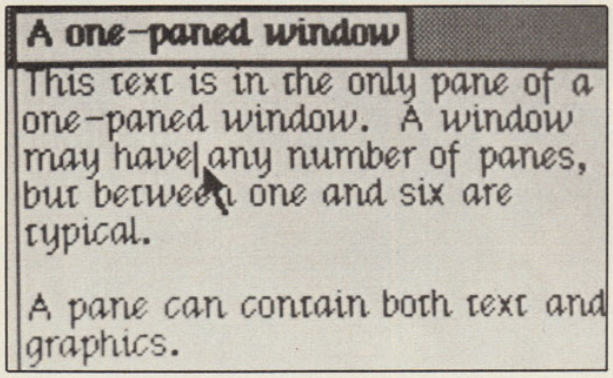


A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

(8b)

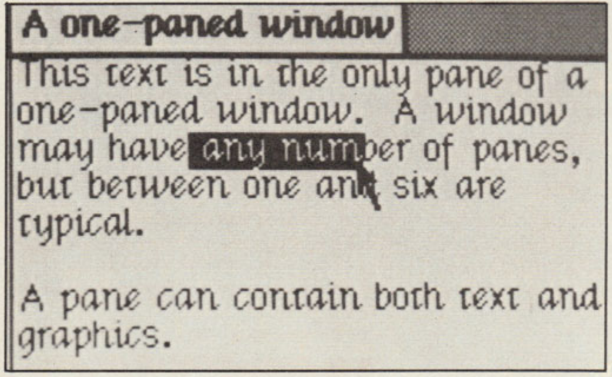


A one-paned window

This text is in the only pane of a one-paned window. A window may have| any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

(8c)



A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

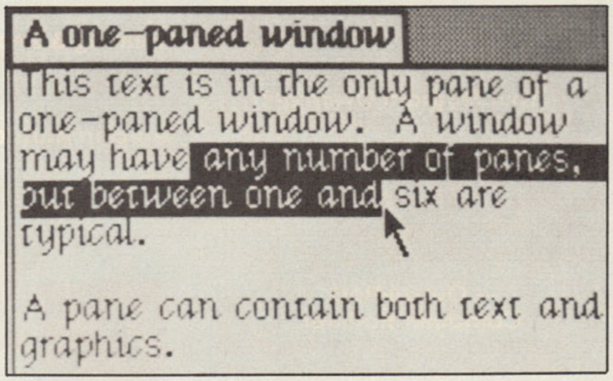
If the button is released without moving the cursor, the bar remains, indicating that a zero-width selection has been made. This method—clicking once between characters—is the one to use before you insert new text.

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 (photo 8c). When the button is released, the selected characters remain highlighted (photo 8d). This method—drawing through a passage—is the one to use before you copy, move, delete, or replace text, or before you change it 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 (photo 8e). This special mechanism is provided because it is very common to select a word. Informal experiments lead us to believe that double clicking is much easier than 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 screen 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*.

(8d)

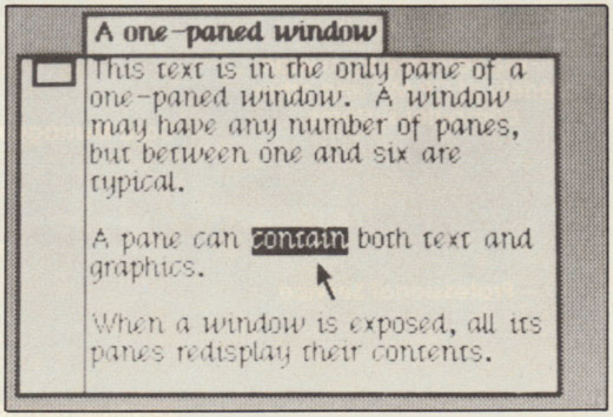


A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

(8e)



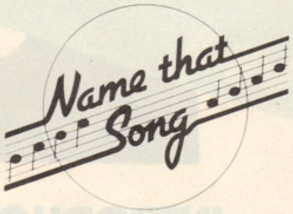
A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can contain both text and graphics.

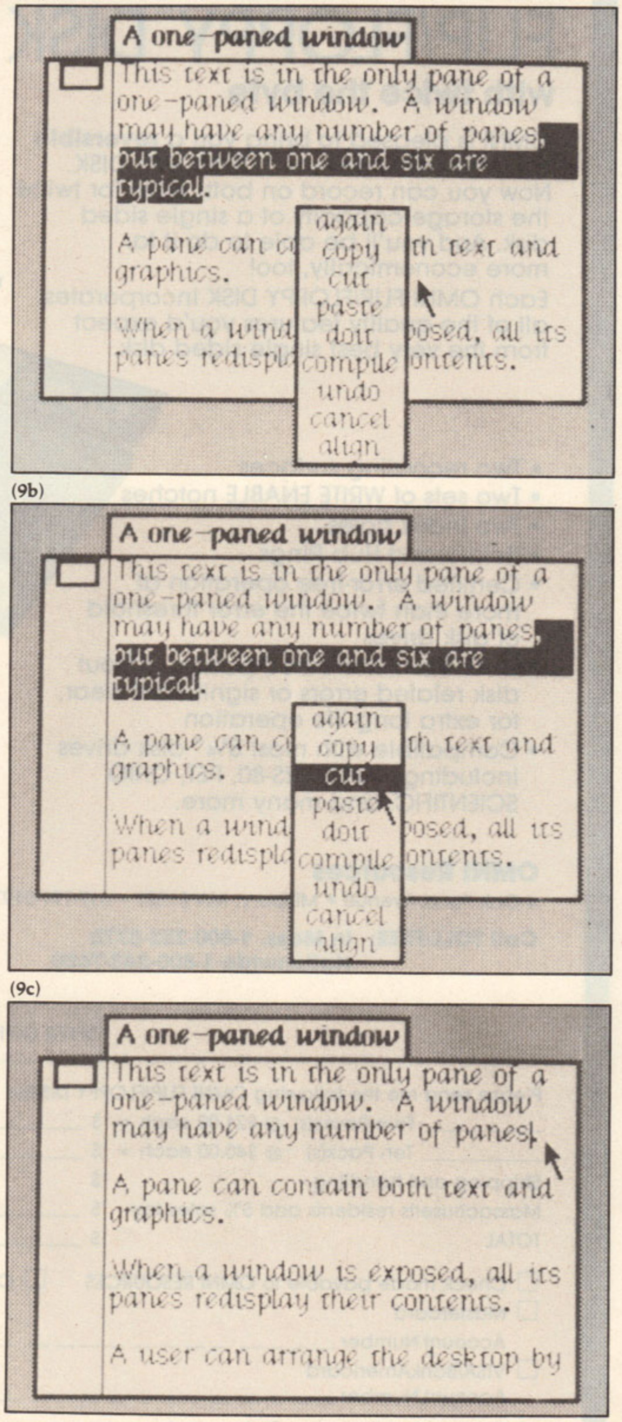
When a window is exposed, all its panes redisplay their contents.

Photo 8: *Selecting text using the mouse and the cursor.*



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.


(9a)

A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can co[ntain bo]th text and graphics.

When a wind[ow is ex]posed, all its panes redispl[ay their c]ontents.

(9b)

A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes, but between one and six are typical.

A pane can co[ntain bo]th text and graphics.

When a wind[ow is ex]posed, all its panes redispl[ay their c]ontents.

(9c)

A one-paned window

This text is in the only pane of a one-paned window. A window may have any number of panes

A pane can contain both text and graphics.

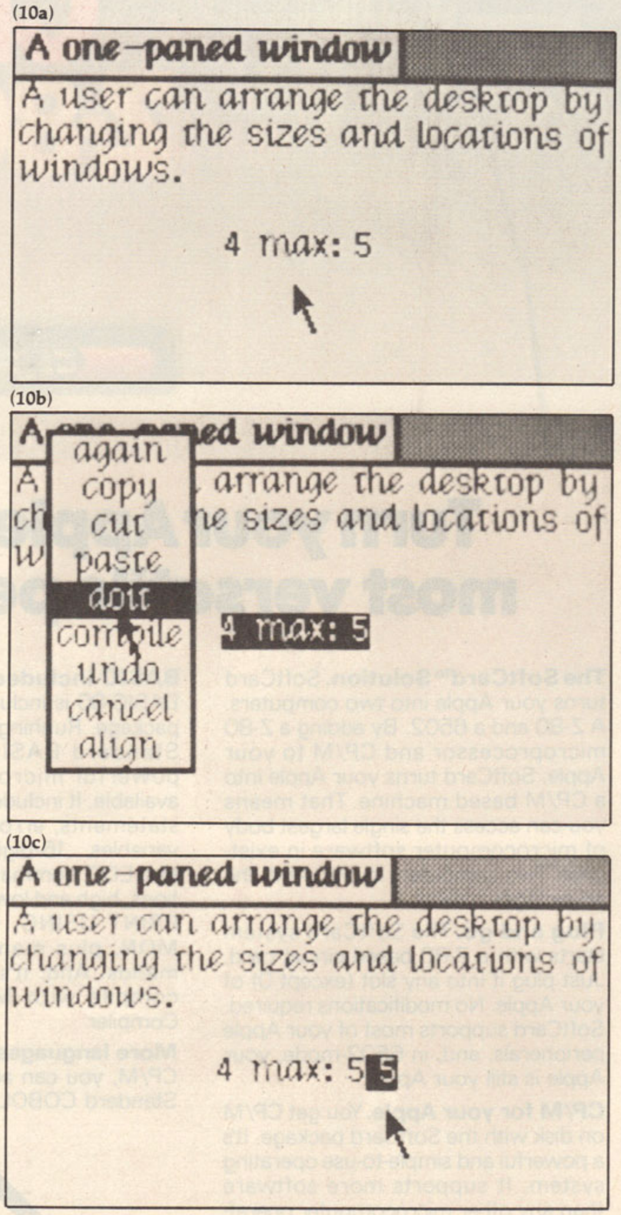
When a window is exposed, all its panes redisplay their contents.

A user can arrange the desktop by

Photo 9: *"Cutting" text in Smalltalk.*

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 (photo 9a), keep that button down while moving the cursor to the command name (photo 9b), and then release the button (photo 9c). 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 statement in the Smalltalk language (photo 10a). Then you select that statement and issue the


(10a)

A one-paned window

A user can arrange the desktop by changing the sizes and locations of windows.

4 max: 5

(10b)

A one-paned window

A user can arrange the desktop by changing the sizes and locations of windows.

4 max: 5

(10c)

A one-paned window

A user can arrange the desktop by changing the sizes and locations of windows.

4 max: 55

Photo 10: *Executing text using the "doit" message.*

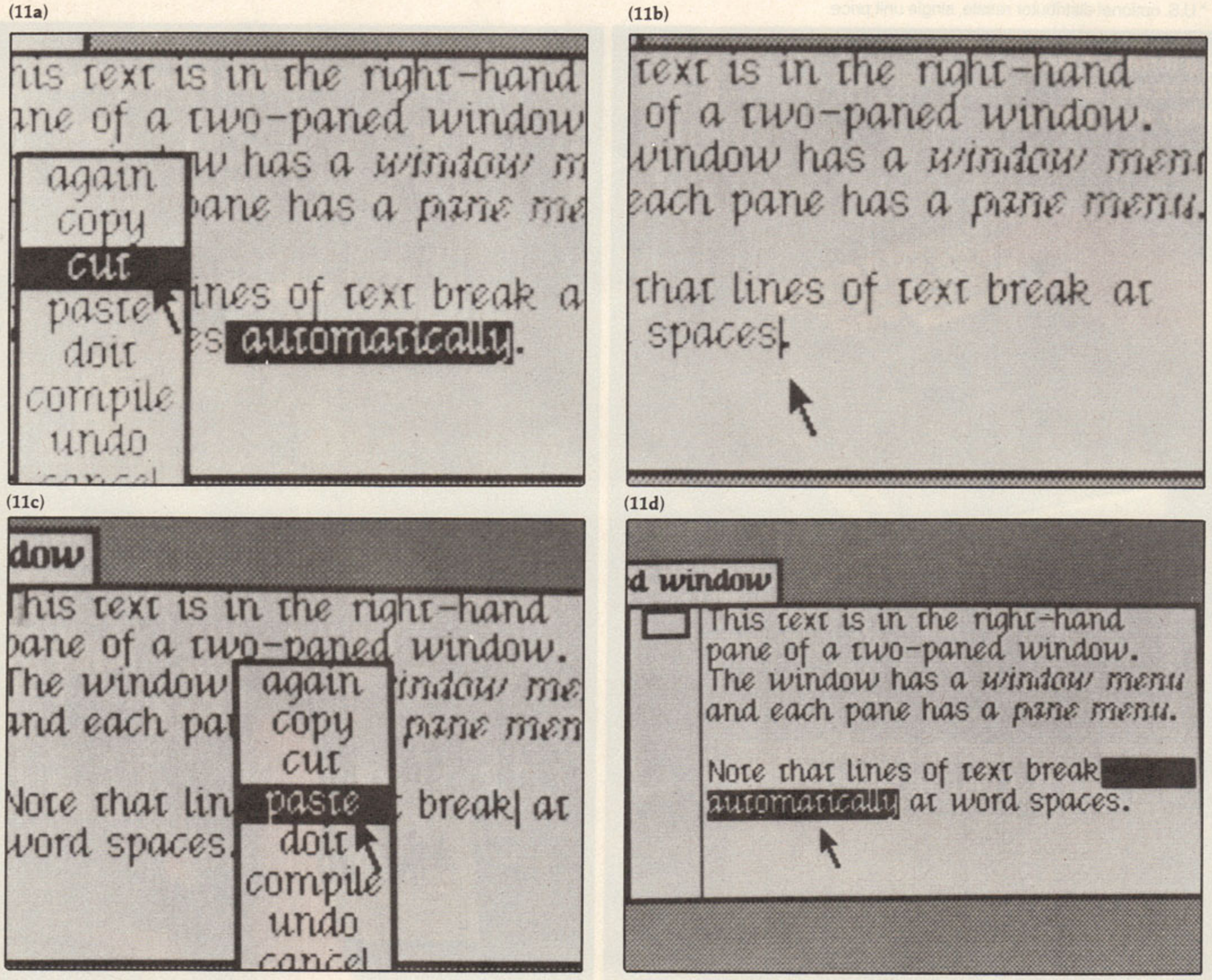
single-parameter command "do it" (photo 10b) to obtain the result (photo 10c). The "do it" command provides immediate execution of any Smalltalk statement or group of statements. This method of command issuance uses the previous method: you are sending the message *doit* to the pane, with the Smalltalk 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 work-space 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 there" 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 messages to more than one pane, the source pane and the destination pane. In a 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 (photo 11a). The "cut" command deletes the selected text (photo 11b), but leaves it in a special place where it can be retrieved by "paste." Then you select the destination and issue the "paste" command (photo 11c) to complete the move (photo 11d).
- A copy is done by *copy and paste*, which is completely analogous to cut and paste, but does not delete the original text.

Remember the "copy-from-file" example (the one where you had to go in and out of many layers of modes)? In the Smalltalk-76 user interface, you can accomplish this with six pushed buttons, no mode exits, 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


(11a)

his text is in the right-hand
ane of a two-paned window
w has a window m
pane has a pane me
ines of text break a
es automatically.

(11b)

text is in the right-hand
of a two-paned window.
window has a window men
each pane has a pane menu.

that lines of text break at
spaces

(11c)

dow

This text is in the right-hand
pane of a two-paned window.
The window
and each pa
Note that lin
word spaces.

(11d)

d window

This text is in the right-hand
pane of a two-paned window.
The window has a window menu
and each pane has a pane menu.

Note that lines of text break automatically at word spaces.

Photo 11: *Moving text in Smalltalk.*

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*, and *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 (table 1) 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 (photo 12a). In a few seconds, a two-paned "inspect window" appears on the screen. Its title 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 (photo 12b), in this case, 3. You point to the word center and click. In the value pane appears the value of center (photo 12c), 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

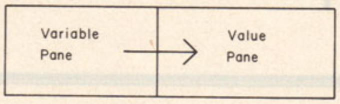


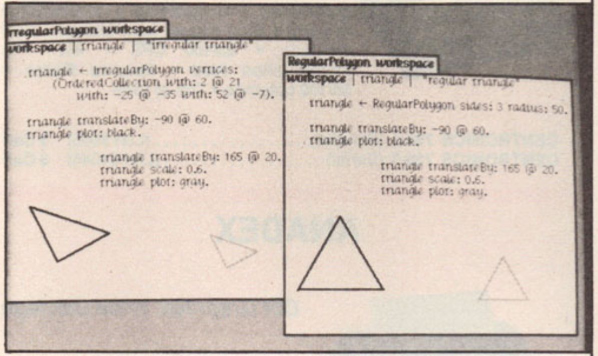
Figure 3: *Principal dependencies among panes of an inspect window.*

The following template contains a description of a regular polygon with the following attributes:

sides	Number of sides (3 for a triangle, 5 for a pentagon, etc.).
center	If the regular polygon were inscribed in a circle, this would be its center point.
radius	If the regular polygon were inscribed in a circle, this would be its radius.
plotter	A pen that can draw an image of the polygon on the screen or on paper.

The following expressions provide an example of creating and using an instance of RegularPolygon.

```
triangle ← RegularPolygon sides: 3 radius: 50.

triangle translateBy: -90 @ 60.
triangle plot: black.

triangle translateBy: 165 @ 20.
triangle scale: 0.6.
triangle plot: gray.          "where gray denotes an ink color"
```


IrregularPolygon workspace
workspace | triangle | "irregular triangle"
RegularPolygon workspace
workspace | triangle | "regular triangle"

```
triangle ← IrregularPolygon vertices:
    (OrderedCollection with: 2 @ 21
        with: -25 @ -35 with: 52 @ -7).

triangle translateBy: -90 @ 60.
triangle plot: black.

        triangle translateBy: 165 @ 20.
        triangle scale: 0.6.
        triangle plot: gray.
```

```
triangle ← RegularPolygon sides: 3 radius: 50.

triangle translateBy: -90 @ 60.
triangle plot: black.

        triangle translateBy: 165 @ 20.
        triangle scale: 0.6.
        triangle plot: gray.
```

class name	RegularPolygon
superclass	Object
instance variable names	sides center radius plotter
class messages and methods	

```
initialization
    sides: s radius: r | |
    "Create an instance of RegularPolygon whose center is located at the center of the currently active window on the display screen. Screen is a
    global variable that refers to the hardware display screen."
    ↑ self new sides: s radius: r center: (Screen activeWindow frame center)
```

instance messages and methods

```
initialization
    sides: s radius: r center: c | |
    "Initialize all attributes. Class Pen is provided in the system as one way of side effecting the display screen."
    sides ← s.
    center ← c.
    radius ← r.
    plotter ← Pen width: 2
analysis
    center | |                                      "Answer the center coordinate of the polygon."
    ↑center
    sides | |                                       "Answer the polygon's number of sides."
    ↑sides
display
    plot: ink | |                                   "Draw an image of the polygon using the specified ink color."
    plotter penup.                                  "lift the pen to disable drawing"
    plotter goto: self center.                      "position the pen at the center"
    plotter up.                                     "face the top of the screen"
    plotter go: radius.                             "position at a corner"
    plotter turn: 180 - (self cornerAngle/2).       "turn to face along a side"
    plotter color: ink.                             "select the ink color"
    plotter pendn.                                  "lower the pen to enable drawing"
    1 to: sides do:                                 "for each side of the polygon:"
        [:i | plotter go: self sideLength.          "plot that side"
            plotter turn: 180-self cornerAngle]     "turn to face along the next side"
transformation
    scale: factor | |
    "Scale the polygon radius by the specified factor."
    radius ← radius * factor
    translateBy: deltaXY | |
    "Change the polygon's location by the specified amount (a Point)."
    center ← center + deltaXY
private instance methods
    cornerAngle | | "Answer the interior angle of any vertex, in degrees."
    ↑180 - (360 / sides)
    sideLength | | "Answer the length of any one of the equal sides."
    ↑2 * radius * (self cornerAngle /2) degreesToRadians cos
```

Table 1: *Description and class template for class* RegularPolygon.

Circle 397 on inquiry card. →

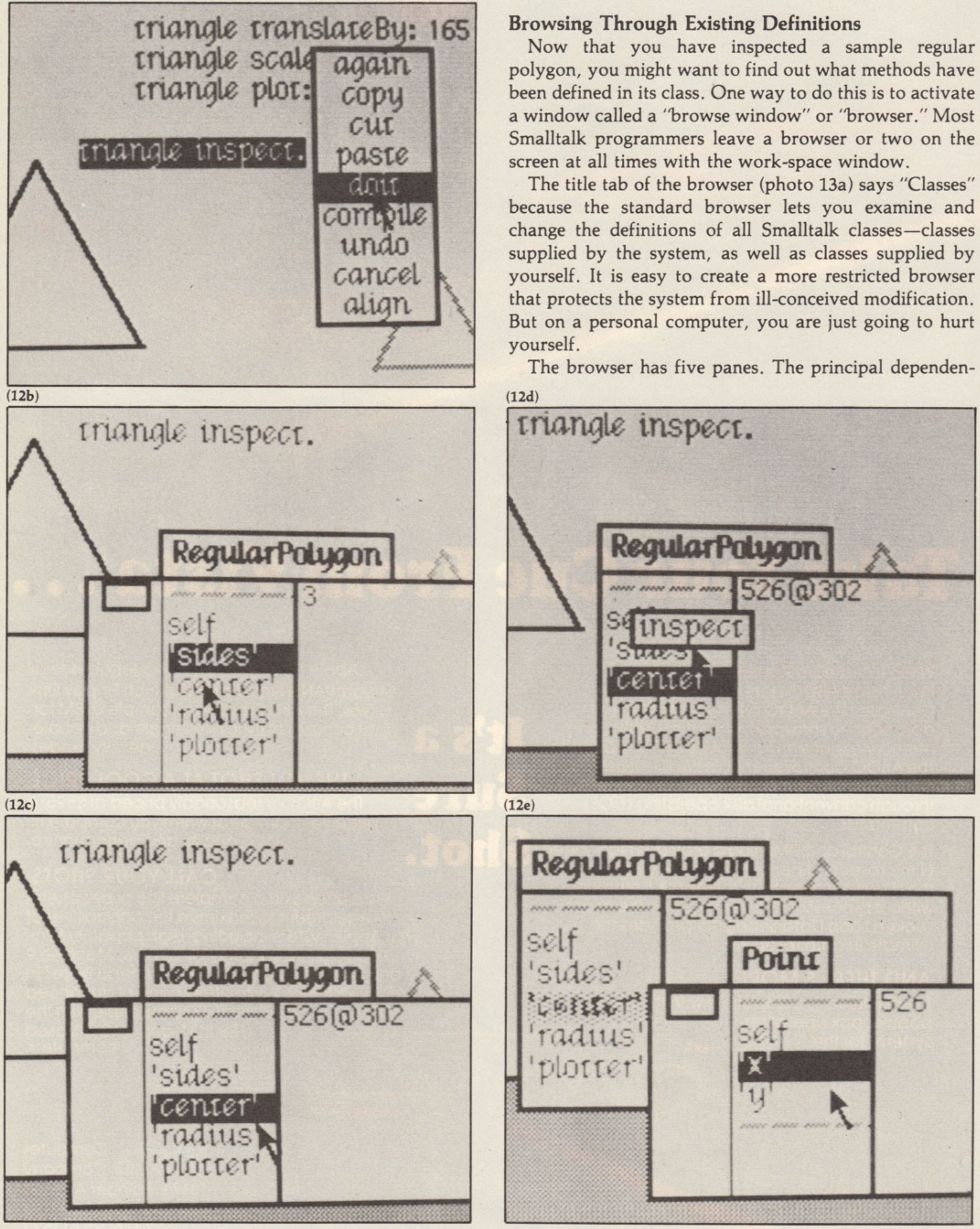
the "inspect" command (photo 12d). On the screen appears another inspect window showing that center is an instance of class Point (photo 12e). You can now examine that point's variables, x and y, reactivate the original inspect window, close either or both windows, or work in any other window. You are not 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 way 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 (photo 13a) 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 dependen-



(12a)

```
triangle translateBy: 165
triangle scal
triangle plot:
triangle inspect.
```

(12b)

```
triangle inspect.
```

(12c)

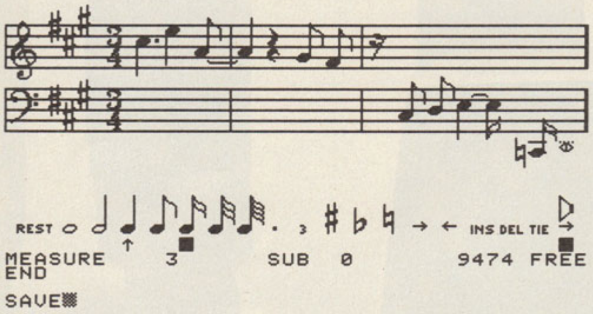
```
triangle inspect.
```

(12d)

```
triangle inspect.
```

(12e)

Photo 12: *Inspecting data structures in Smalltalk.*



cies 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 is 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 to 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 of 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 class 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 method, translateBy:. Click its name in the method pane and its definition is immediately displayed in that pane's dependent, the editing pane (photo 13b). 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 (photo 13c).

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 (photo 13d).

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 (photo 13e). 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

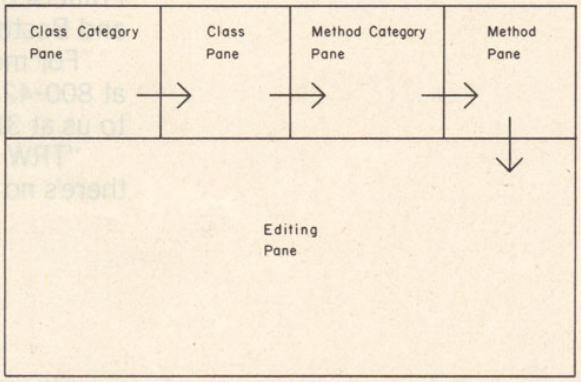


Figure 4: *Principal dependencies among panes of a browse window.*

the desired class (photo 13f).

Categorization 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.

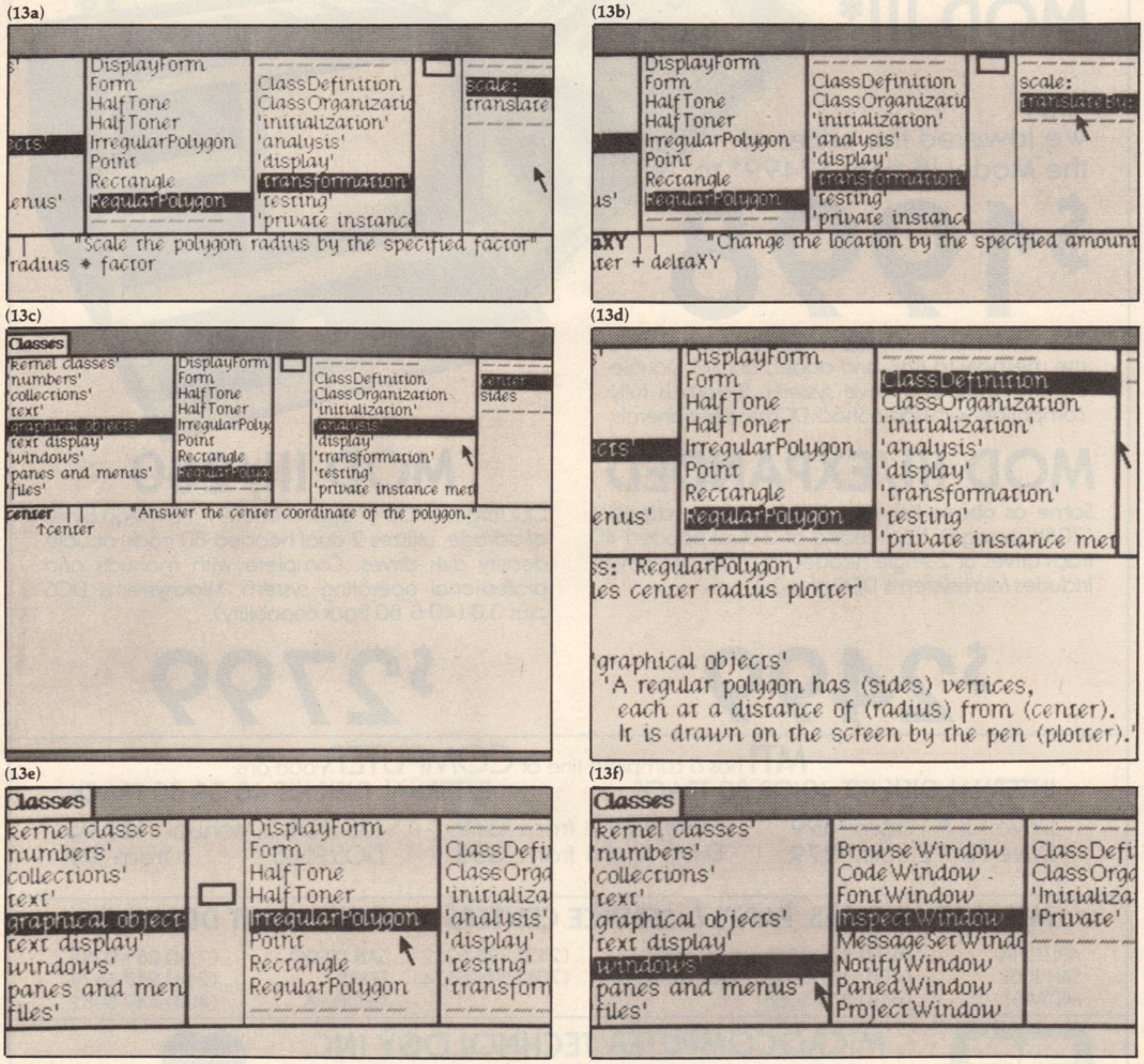
If you just want to browse around reading class and method definitions, you can do so by lazily clicking the selection button with the cursor over each name, never touching the keyboard. That is why the window is called a browser. Browsers are further discussed in references 6 and 7.

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 any 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.



(13a)

```
 |      "Scale the polygon radius by the specified factor"
radius * factor
```

(13b)

```
aXY | |      "Change the location by the specified amount
ter + deltaXY
```

(13c)

```
center | |      "Answer the center coordinate of the polygon."
    ↑center
```

(13d)

```
ss: 'RegularPolygon'
les center radius plotter'


'graphical objects'
 'A regular polygon has (sides) vertices,
  each at a distance of (radius) from (center).
  It is drawn on the screen by the pen (plotter).'
```

(13e)

(13f)

Photo 13: *Browsing through existing definitions in Smalltalk.*

The following template contains a description of an irregular polygon with the following attributes:

vertices	An OrderedCollection of Points.
plotter	A pen that can draw an image of the polygon on the screen or on paper.

The following expressions provide an example of creating and using an instance of IrregularPolygon.

```
triangle ← IrregularPolygon vertices:
                (OrderedCollection with: 2 @ 21
                                   with: -25 @ -35
                                   with: 52 @ -7).
triangle translateBy: -90 @ 60.
triangle plot: black.

triangle translateBy: 165 @ 20.
triangle scale: 0.6.
triangle plot: gray.  "where gray denotes an ink color"
```

class name	IrregularPolygon
superclass	Object
instance variable names	vertices plotter

class messages and methods

```
  initialization
    vertices: aCollection | |
      "Create an instance of IrregularPolygon whose center is located at the center of the currently active window on the display screen.
      Screen is a global variable that refers to the hardware display screen."
      ↑ self new vertices: aCollection center: (Screen activeWindow frame center)
```

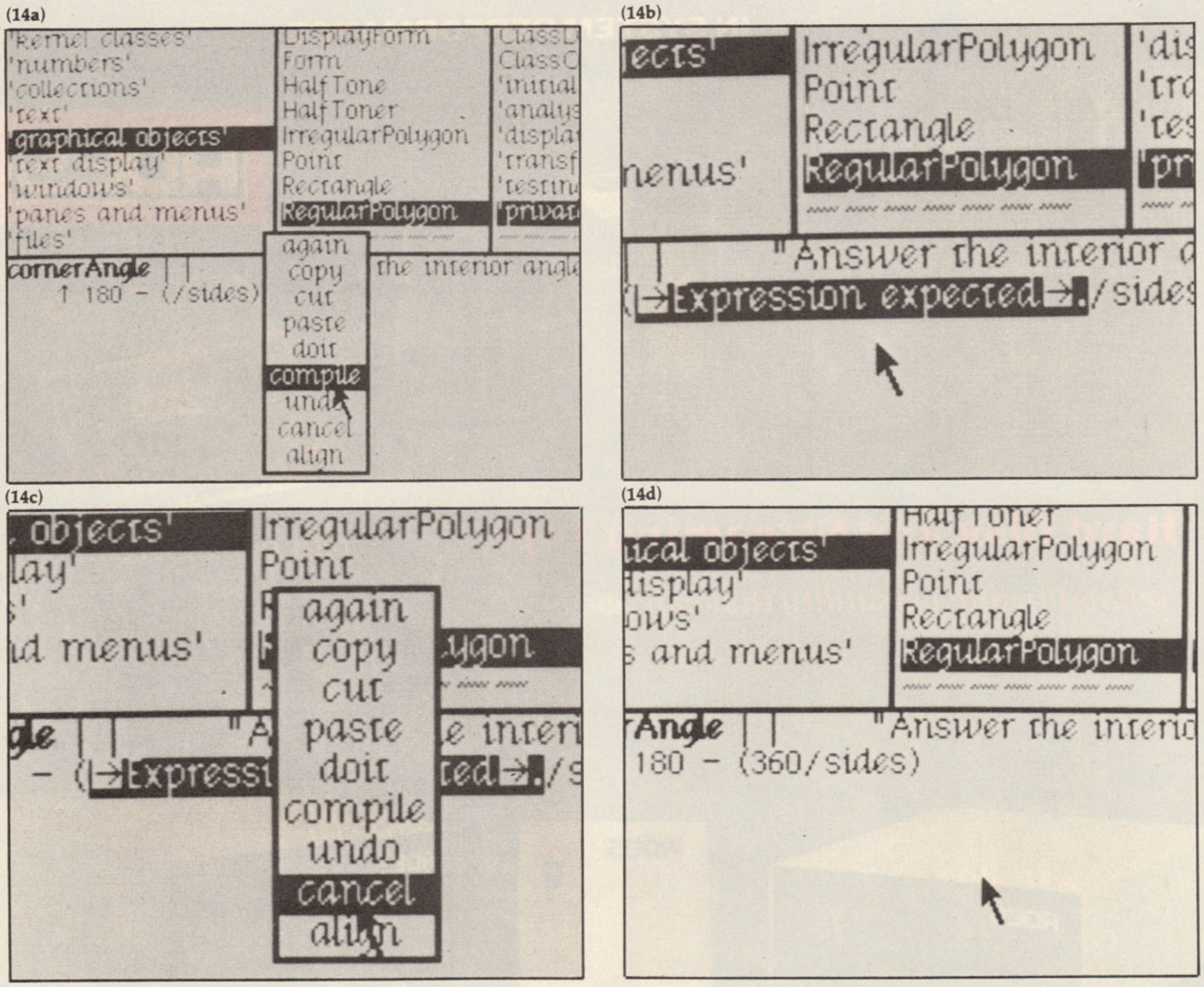
instance messages and methods

```
  initialization
    vertices: aCollection center: c | |
      "Initialize all attributes. Class Pen is provided in the system as one way of side effecting the display screen."
      vertices ← aCollection.
      plotter ← Pen width: 2.
      self translateBy: c - self center
  analysis
    center | sum |                              "Answer the center coordinate of the polygon."
      sum ← 0@0.
      vertices do: [ :pt | sum ← sum + pt].
      ↑sum / self sides
    sides | |                                   "Answer the polygon's number of sides."
      ↑vertices size
  display
    plot: ink | |                               "Draw an image of the polygon using the specified ink color."
      plotter penup.                            "lift the pen to disable drawing"
      plotter goto: vertices last.              "position the pen at one vertex"
      plotter color: ink.                       "select the ink color"
      plotter pendn.                            "lower the pen to enable drawing"
      vertices do:                              "for each vertex"
        [:pt | plotter goto: pt]                "draw a straight line to it"
  transformation
    scale: factor | center |
      "Scale the polygon by the specified factor."
      center ← self center.                     "the center of expansion"
      vertices ← vertices collect:              "generate new vertex list from old list"
        [ :pt | (pt - center)*factor + center]
    translateBy: deltaXY | |
      "Change the polygon's location by the specified amount (a Point)."
      vertices ← vertices collect: [:vertex | vertex + deltaXY]
```

Table 2: *Description and class template for class* IrregularPolygon.

When you are done editing, pop up the active-pane menu and issue the "compile" command (photo 14a). 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 (photo 14b). 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 menu and issue the


(14a)

```
cornerAngle || the interior angle
  ↑ 180 - (/sides)
```

(14b)

```
"Answer the interior a
(|→Expression expected→|/sides
```

(14c)

(14d)

```
Angle ||  "Answer the interio
 180 - (360/sides)
```

Photo 14: *Options during method compilation.*

"cancel" command (photo 14c). The "cancel" command redisplays the last successfully compiled version of the method (photo 14d). If you cancel by accident, just issue the "undo" command 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 (photo 15a). 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 (photo 15b).

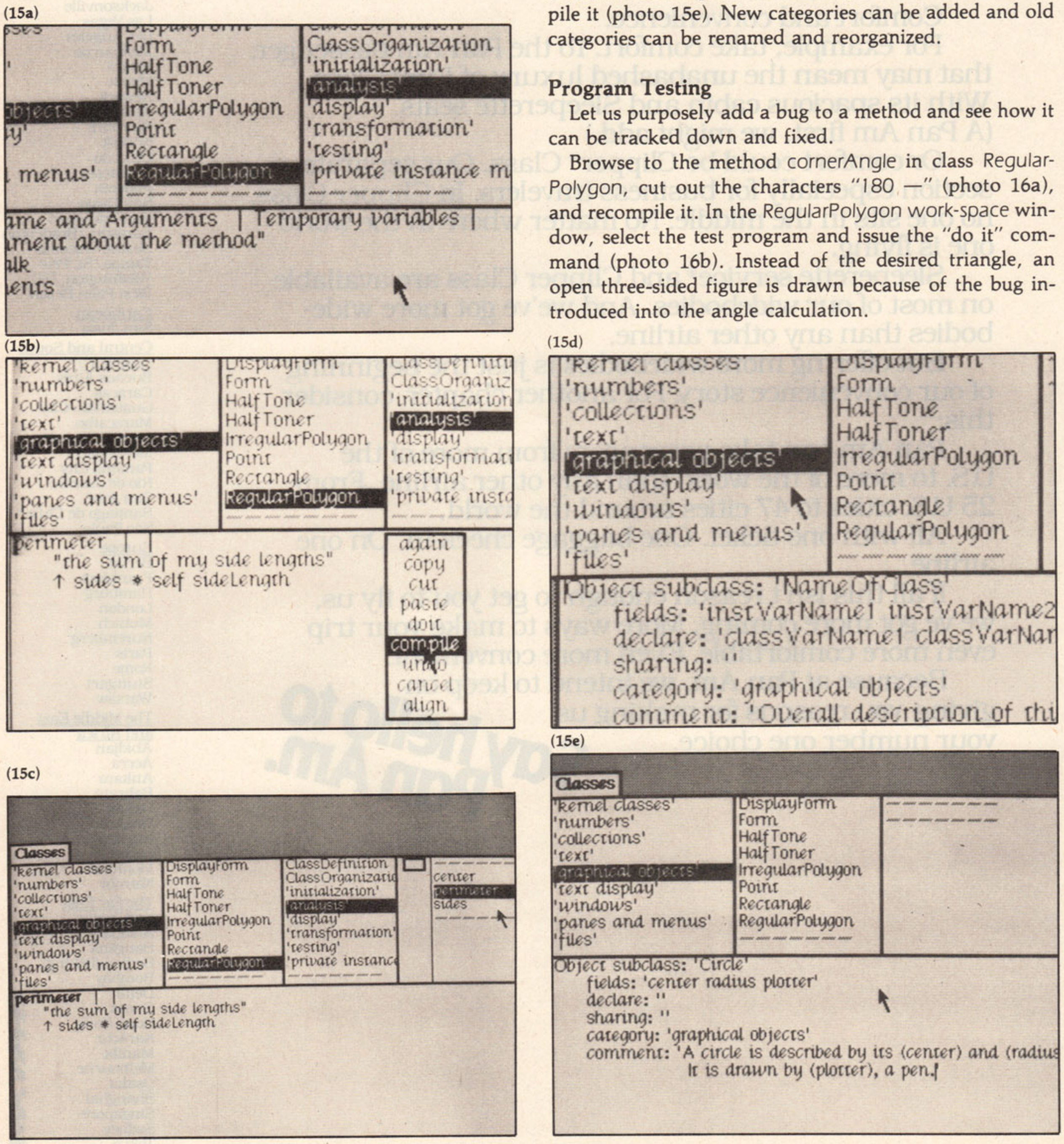
Once compilation succeeds, the selector of the new method is automatically added to the alphabetized list in the method pane, and the message pattern is automatically changed to boldface in the editing pane (photo 15c).

A new class definition is added in an analogous manner. Start by selecting a class category (photo 15d), then fill in a template for defining a new class and compile it (photo 15e). New categories can be added and old categories can be renamed and reorganized.

Program Testing

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

Browse to the method cornerAngle in class RegularPolygon, cut out the characters "180 —" (photo 16a), and recompile it. In the RegularPolygon work-space window, select the test program and issue the "do it" command (photo 16b). Instead of the desired triangle, an open three-sided figure is drawn because of the bug introduced into the angle calculation.



(15a)

```
ame and Arguments | Temporary variables |
ment about the method"
alk
ents
```

(15b)

```
perimeter | |
    "the sum of my side lengths"
    ↑ sides * self sideLength
```

(15c)

```
perimeter
    "the sum of my side lengths"
    ↑ sides * self sideLength
```

(15d)

```
Object subclass: 'NameOfClass'
    fields: 'instVarName1 instVarName2
    declare: 'classVarName1 classVarNar
    sharing: ''
    category: 'graphical objects'
    comment: 'Overall description of thi
```

(15e)

```
Object subclass: 'Circle'
    fields: 'center radius plotter'
    declare: ''
    sharing: ''
    category: 'graphical objects'
    comment: 'A circle is described by its (center) and (radius
              It is drawn by (plotter), a pen.'
```

Photo 15: *Adding new definitions in Smalltalk.*

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 (photo 16c). Now rerun the test case. When the computer encounters the breakpoint, a new window appears in midscreen. It is called a "notify window" (photo 16d). The title tab of the notify window says "about to calculate angle".

The notify window has 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" (photo 16e).

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.

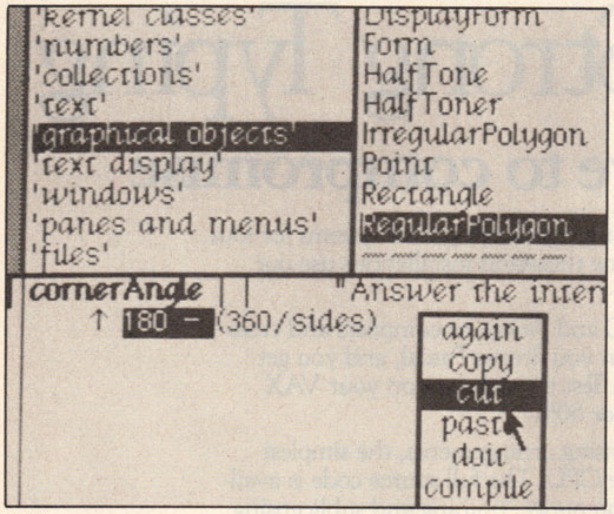
What a Notify Window Can Display

The "stack" command expands the contents of the pane to include messages that have been sent, but have not yet received replies (photo 17a). 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 (photo 17b). If "close" were issued, the notify window would disappear from the screen and execution of the

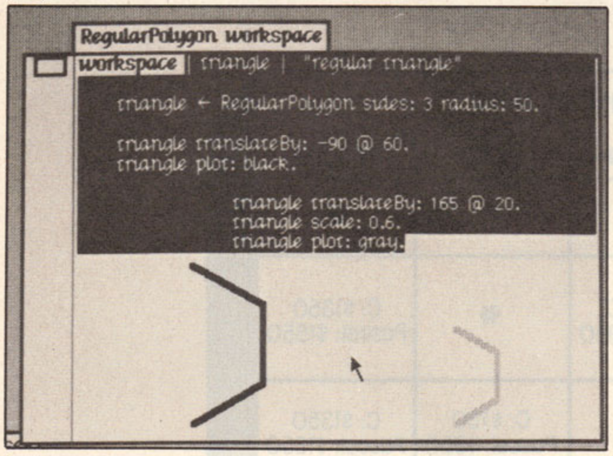
Text continued on page 138

(16a)



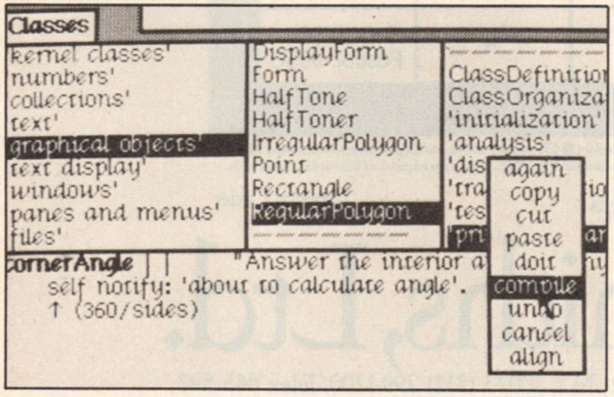
```
cornerAngle | |         "Answer the inter
    ↑ 180 - (360/sides)
```

(16b)



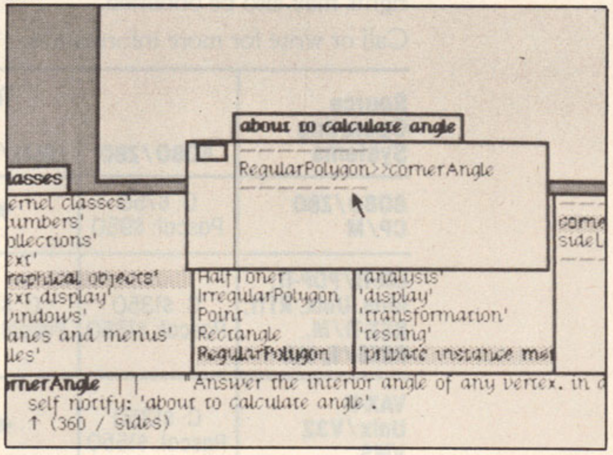
```
| triangle |  "regular triangle"

    triangle ← RegularPolygon sides: 3 radius: 50.

    triangle translateBy: -90 @ 60.
    triangle plot: black.

                triangle translateBy: 165 @ 20.
                triangle scale: 0.6.
                triangle plot: gray.
```

(16c)



```
cornerAngle | |     "Answer the interior a
    self notify: 'about to calculate angle'.
    ↑ (360/sides)
```

(16d)



```
rnerAngle | |     "Answer the interior angle of any vertex. in a
    self notify: 'about to calculate angle'.
    ↑ (360 / sides)
```

(16e)

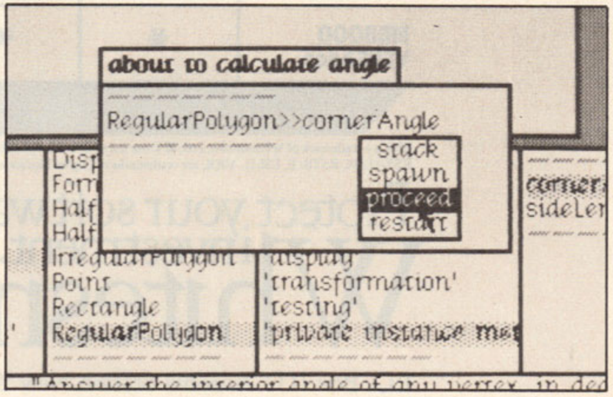


Photo 16: *Creating a faulty method for purposes of illustration.*

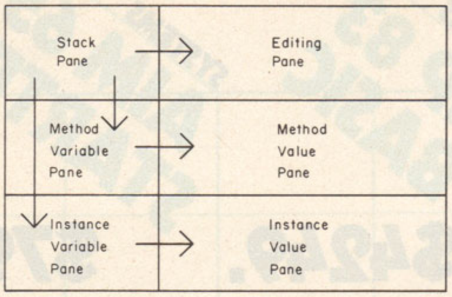


Figure 5: *Principal dependencies among panes of a notify window.*

Text continued from page 134:

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 (photo 17c). 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.

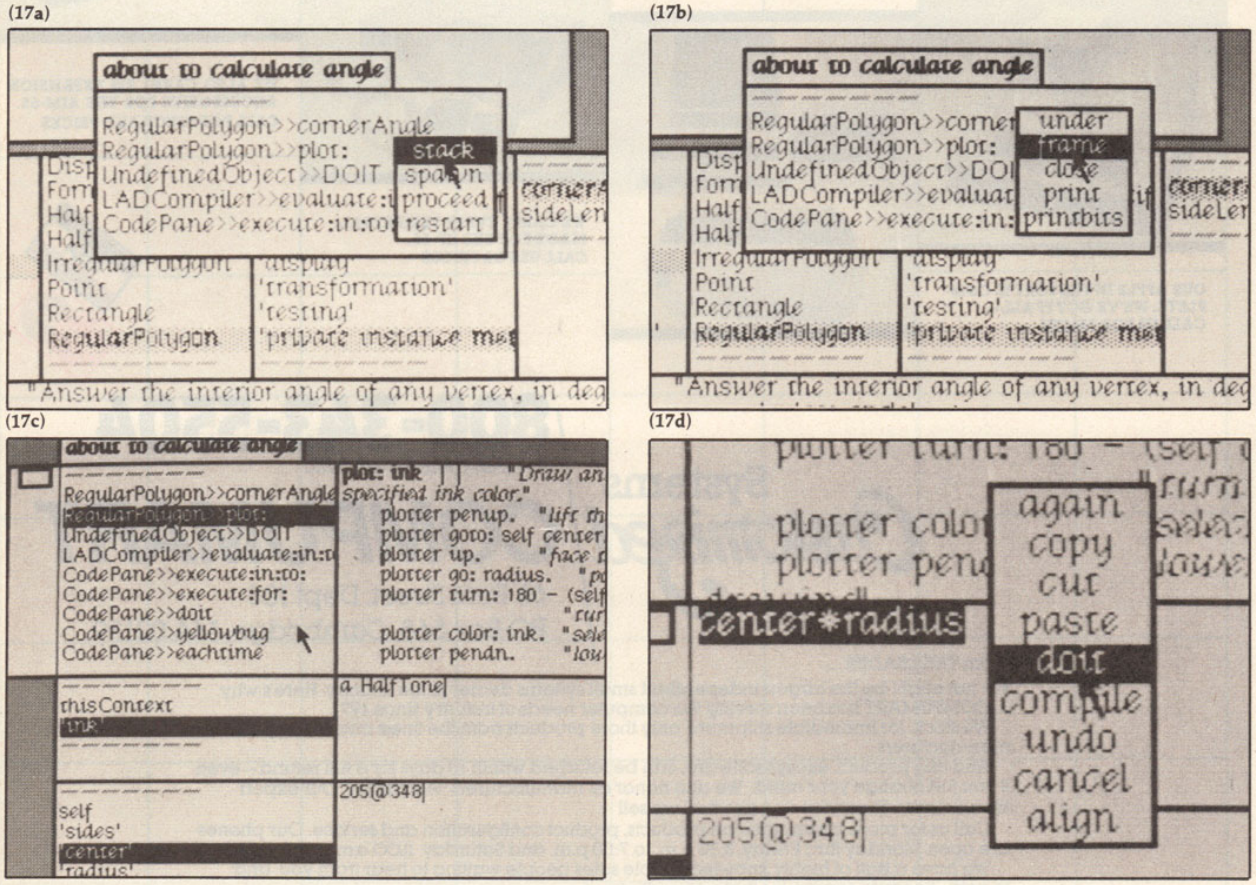
The middle two panes are the "context variable" and "context value" panes. They are analogous to the two panes of an inspect window, but, in this case, the variables you can examine are the arguments and local variables of the method selected in the stack pane. Click ink in the variable pane to see its 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" (photo 17d). 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 (photo 18a). The answer, 120, appears to the right of the question (photo 18b). Since the interior angle of a regular


(17a)

```
about to calculate angle
RegularPolygon>>cornerAngle
RegularPolygon>>plot:
UndefinedObject>>DOIT
LADCompiler>>evaluate:i
CodePane>>execute:in:to:
Disp  Form  Half  Half
IrregularPolygon   'display'
Point              'transformation'
Rectangle          'testing'
RegularPolygon     'private instance me
cornerA  sideLen
"Answer the interior angle of any vertex, in deg
```

(17b)

```
about to calculate angle
RegularPolygon>>corner
RegularPolygon>>plot:
UndefinedObject>>DOI
LADCompiler>>evaluat
CodePane>>execute:in:
Disp  Form  Half  Half
IrregularPolygon   'display'
Point              'transformation'
Rectangle          'testing'
RegularPolygon     'private instance me
corner  sideLen
"Answer the interior angle of any vertex, in deg
```

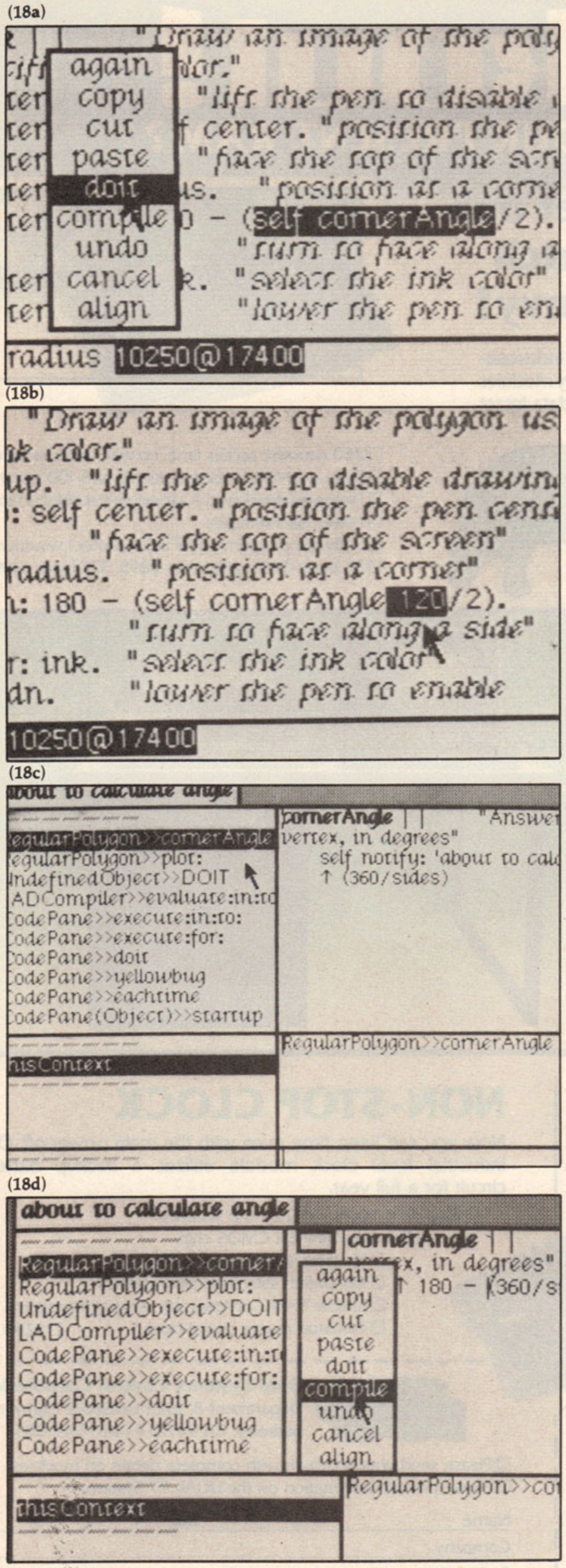
(17c)

```
about to calculate angle
RegularPolygon>>cornerAngle    plot: ink      "Draw an
RegularPolygon>>plot:          specified ink color."
UndefinedObject>>DOIT              plotter penup.  "lift th
LADCompiler>>evaluate:in:t         plotter goto: self center.
CodePane>>execute:in:to:           plotter up.     "face n
CodePane>>execute:for:             plotter go: radius.  "p
CodePane>>doit                     plotter turn: 180 - (self
CodePane>>yellowbug                                "tur
CodePane>>eachtime                 plotter color: ink.  "sele
                                   plotter pendn.   "low
thisContext                    a HalfTone
ink
self                           205@348
'sides'
center
'radius'
```

(17d)

```
plotter turn: 180 - (self
                      "turn
plotter colo          "selec
plotter pen           "lowe
center*radius
205@348
```

Photo 17: *Use of the "notify" window.*



(18a)

```
  "Draw an image of the poly
          lor."
          "lift the pen to disable
        f center. "position the pe
          "face the top of the scr
        us.  "position at a corne
        0 - (self cornerAngle/2).
          "turn to face along a
        k.  "select the ink color"
          "lower the pen to en
radius 10250@17400
```

(18b)

```
  "Draw an image of the polygon us
nk color."
up.  "lift the pen to disable drawin
: self center. "position the pen cent
      "face the top of the screen"
radius.  "position at a corner"
n: 180 - (self cornerAngle 120/2).
         "turn to face along a side"
r: ink.  "select the ink color"
dn.  "lower the pen to enable
10250@17400
```

(18c)

```
RegularPolygon>>cornerAngle
RegularPolygon>>plot:
UndefinedObject>>DOIT
LADCompiler>>evaluate:in:to
CodePane>>execute:in:to:
CodePane>>execute:for:
CodePane>>doit
CodePane>>yellowbug
CodePane>>eachtime
CodePane(Object)>>startup
```

```
cornerAngle | |     "Answer
vertex, in degrees"
    self notify: 'about to cal
    ↑ (360/sides)
```

```
thisContext
RegularPolygon>>cornerAngle
```

(18d)

```
RegularPolygon>>cornerA
RegularPolygon>>plot:
UndefinedObject>>DOIT
LADCompiler>>evaluate
CodePane>>execute:in:t
CodePane>>execute:for:
CodePane>>doit
CodePane>>yellowbug
CodePane>>eachtime
```

```
cornerAngle | |
vertex, in degrees"
    ↑ 180 - (360/s
```

```
thisContext
RegularPolygon>>co
```

Photo 18: *Debugging a faulty method.*

triangle is 60 degrees, we have found the planted bug.

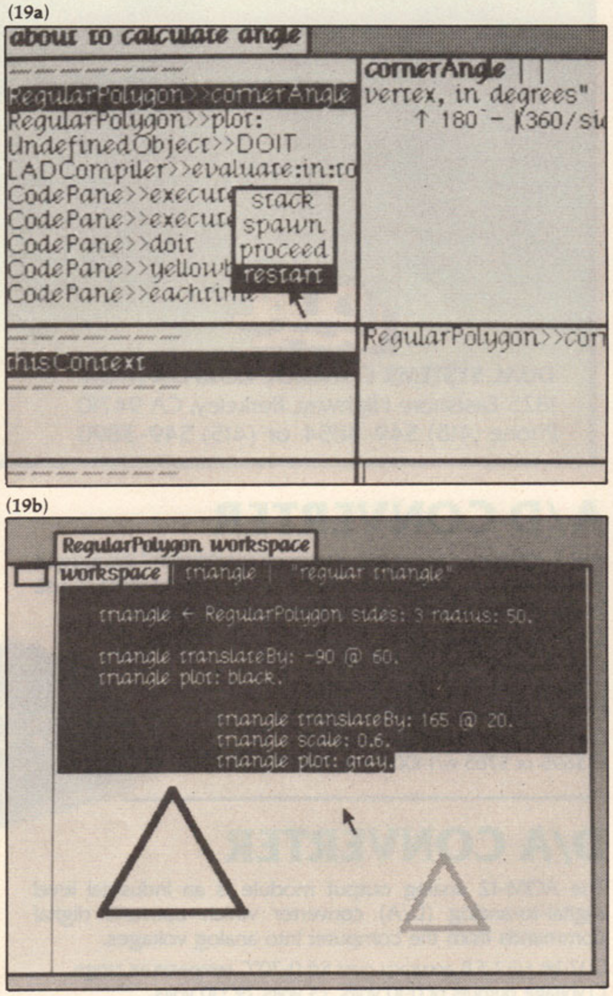
Now select RegularPolygon>>cornerAngle in the stack pane. Its method definition, including the breakpoint we set, appears in the editing pane (photo 18c). Use standard editing to remove the breakpoint, correct the error, and recompile the editing pane (photo 18d).

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 (photo 19a). This lets the test proceed (photo 19b) without having to start over from the work-space window. Resumption of execution after a correction is a handy capability when a program that has been running 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



(19a)

```
RegularPolygon>>cornerAngle
RegularPolygon>>plot:
UndefinedObject>>DOIT
LADCompiler>>evaluate:in:to
CodePane>>execut
CodePane>>execut
CodePane>>doit
CodePane>>yellowb
CodePane>>eachtim
```

```
cornerAngle | |
vertex, in degrees"
    ↑ 180 - (360/si
```

```
thisContext
RegularPolygon>>cor
```

(19b)

```
triangle |  "regular triangle"

triangle ← RegularPolygon sides: 3 radius: 50.

triangle translateBy: -90 @ 60.
triangle plot: black.

          triangle translateBy: 165 @ 20.
          triangle scale: 0.6.
          triangle plot: gray.
```

Photo 19: *Compilation of a faulty method can be continued without restarting, once the error has been corrected.*

Circle 284 on inquiry card. →

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 something else. There are no modes.

Error Notifications

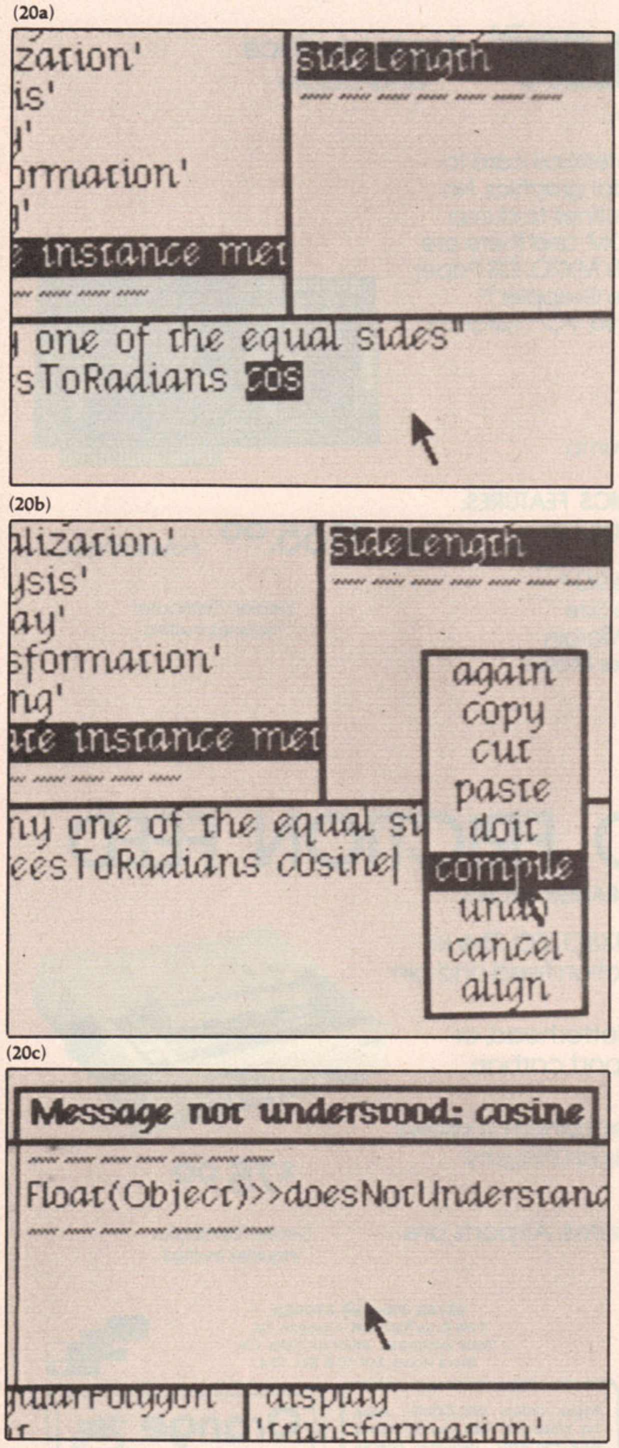
Error messages are no different from breakpoints, except that if they are supposed to be "unrecoverable" they are programmed as:

```
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 neither that object's class nor any of its superclasses defines a method to receive that message. Let us edit the method sideLength (photo 20a) to send the message cosine instead of cos. After recompiling that method (photo 20b) and reexecuting the test program, a notify window appears (photo 20c) 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 run 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 did define a method of that name.


(20a)

```
one of the equal sides"
sToRadians cos
```

(20b)

```
ny one of the equal si
eesToRadians cosine
```

(20c)

Message not understood: cosine

```
Float(Object)>>doesNotUnderstand
```

Photo 20: *Displaying an error in a faulty method.*

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 well-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.

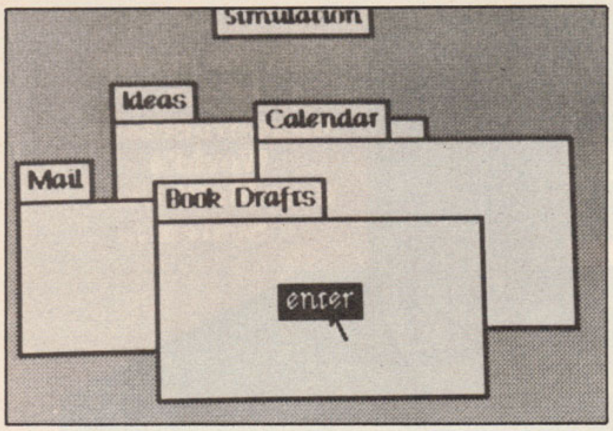
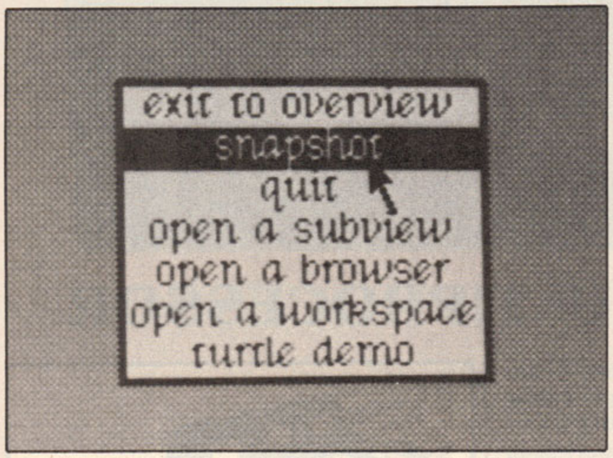


Photo 21: *Project windows in Smalltalk. Each window, when selected, makes available all the windows associated with that project.*

(22a)



(22b)

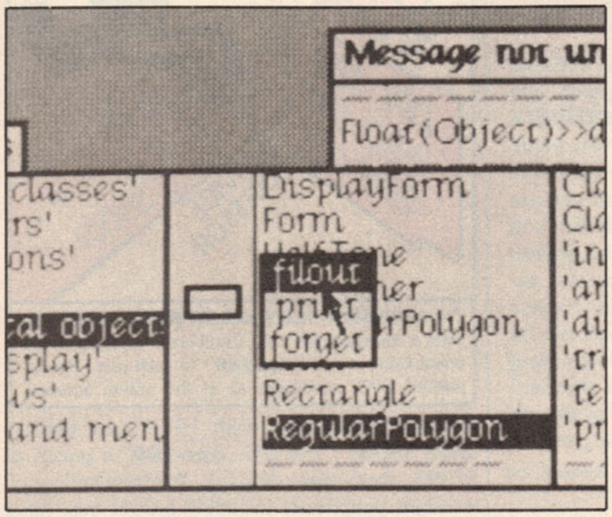


Photo 22: *Recording results in Smalltalk. The current state of the Smalltalk system can be saved with "snapshot." Smalltalk code can be saved to a text file by using "filout" and restored by using "filin."*

- Most compilers can generate more efficient object code if types are declared. Existing implementations of Smalltalk cannot take advantage 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 each 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 (photo 21).

Saving Programs

In unintegrated systems, you create a program using standard text-editing facilities. Then, using standard utility programs, you can obtain 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. Some of the program-saving capabilities of Smalltalk are described briefly below.

One important facility is the *snapshot* (photo 22a). The entire state of the Smalltalk 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 to be listed on a printer. A related facility is *filin/filout*. The *filout* message (photo 22b) writes an ASCII representation of one or more definitions onto a conventional text file. The definitions can then be transfused into another Smalltalk environment 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 the 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

Text continued on page 147

Text continued from page 144:

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 Smalltalk-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 organization is that tasks, including software-development 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 of 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).

User-Oriented Descriptions of Smalltalk Systems

Trygve M H Reenskaug
Central Institute for Industrial Research
Blindern, Oslo 3
Norway

For many people, the workings of a computer remain a mystery. Just exactly what the computer does and how it does it is locked within the code of a computer language. The computer and the user understand two completely different languages. It is well known that only a few systems are designed and written so that they can be understood by the user. More than twenty years of experience has shown that a bad system design can never be hidden from the user, even by a masterfully devised user interface. A quality system, therefore, must be based on sound design that can be described in terms with which the user is familiar.

The Smalltalk system has been designed to handle a great variety of problems and solutions. It, therefore, provides the greatest possible flexibility for writing any kind of system a programmer may desire. While this flexibility is essential for experimenting, there is the potential for disastrous results if restrictions are not put on the system structures that are available to the application programmer.

This article shows how the basic metaphors of Smalltalk can be used to describe complex systems. Since this magazine is not yet distributed in a form readable by Smalltalk, we have to restrict ourselves to traditional written documentation. (Let it be a challenge to Smalltalk experimenters to convert this presentation into a graphic and dynamic one.)

The Smalltalk system user will most likely employ his system to organize the large amount of information that will be available to him, such as reference materials in the form of market information, news services, and weather forecasts. Some data, such as travel information and bank transactions, may flow both to and from the owner. Other information, such as personal notes or material that is not yet ready for distribution, can remain private.

More than twenty years of experience has shown us that a bad system design can never be hidden from the user, even by a masterfully devised user interface.

An individual's total information needs are very large and complex. His Smalltalk system, therefore, is also likely to be large and complex. The challenge to the Smalltalk experimenter is to find ways to structure systems so the user will not only understand how to use them, but also get an intuitive feel for their inner workings. In this way, the user can really be the master and the systems his faithful slaves.

An important part of any system is the software that controls the user's interaction with the information. Mastering the software is crucial to handling the information. With Smalltalk, software is just a special kind of information and is treated as any other information within the total system. It is available to the user in the usual manner.

A traditional way of describing software is through written documentation. Smalltalk provides more dynamic interfaces through the use of two-dimensional graphics and animation on the computer screen. Devising such interfaces is probably the greatest challenge in personal computing today, and it provides a rich field of endeavor for the interested experimenter.

System Descriptions

We can describe any application system in three different ways: *how it is used*, *its system structure*, and *its implementation*:

- ●*How it is used*—This is the least satisfactory type of description. The user operates the system through rote command sequences such as: *switch on the machine, type your password, hit button A, listen to your system saluting you by playing "Hail to the Chief."* Since 80% of all user manuals for electronic data processing systems are of this kind, we will not discuss them further here.

This level of understanding has been likened to walking around in a strange city following directions such as: "Go outside, turn right, walk straight ahead for four blocks, turn left . . . . " It is easy to get lost under such circumstances.

● *System structure*—With this type of description, the user has an intuition about the kinds of building blocks that make up the system, how they behave, and how they interact to form the complete system. We show that the basic Smalltalk metaphors of *objects* and *messages* are well suited to function as building blocks. The metaphors are simple and easy to understand; yet they permit construction of immensely powerful systems.

A basic system will have several thousand objects, and typical applications would contain many more.

Any Smalltalk system contains a large number of objects. A basic system will have several thousand objects, and typical applications would contain many more. The common software engineering device of *layering* becomes essential in making the whole thing manageable. In the description of a layer, essential function on that level is highlighted and inconsequential detail is relegated to lower levels. There is one absolute requirement of these simplified descriptions appearing on the different layers: *what is shown should be correct and complete as far as it goes.* This means that the structure of the description has to be a pure tree structure: the function of each module has to be limited to that module with *no hidden side effects upon the other modules.*

This level of understanding corresponds to the user having a street map of the system. He knows the major landmarks and the most important streets. This gives the user an intuition about the total structure and permits him tc find his way anywhere. It is almost impossible to get totally lost under these circumstances.

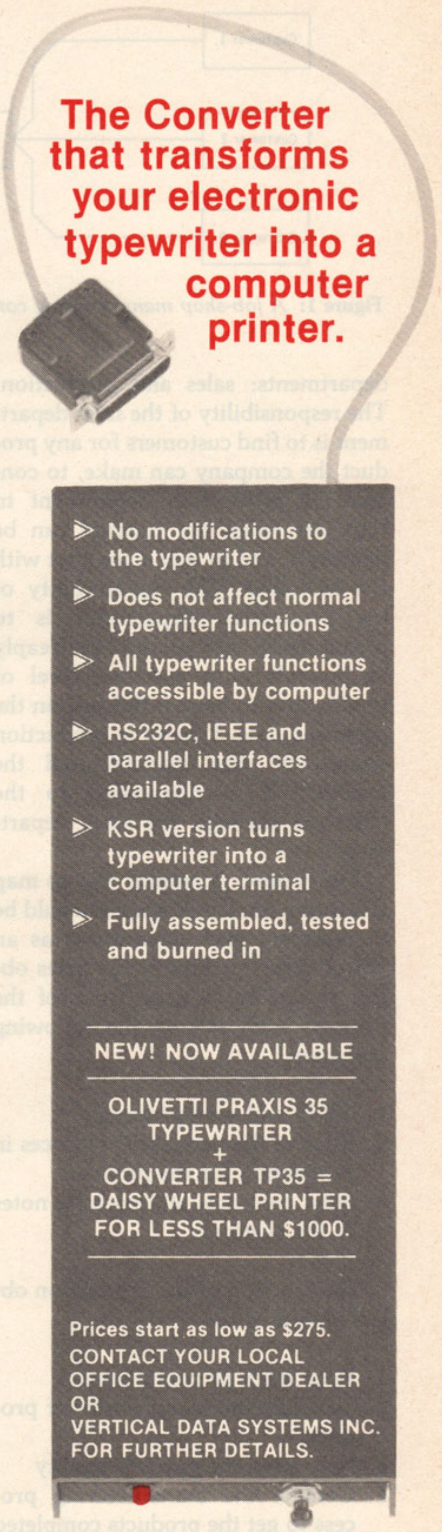
● *Implementation*—Descriptions at this level of understanding explain to the user how each individual object is built so that it behaves in the manner prescribed on the system structure level. Here he will find the third basic metaphor of Smalltalk, the *method.* A method is similar to a subroutine in other languages; it prescribes the actions to be taken by an object when it receives a message.

On all layers but the lowest, the behavior of an object is fairly complex, and we can think of it as composed of a number of *sub-objects* that are used to implement it. The purpose of the method is to enlist the aid of the sub-objects to implement the desired behavior. The user thus finds that the typical object is structured in much the same manner as his total system, and it consists of a number of sub-objects that send messages to each other. The description tool is recursive in that the same tool is used on all levels. This recursion description is probably the most powerful feature of Smalltalk. Once the user masters the few very general concepts, he can learn more and more about his system by simply using these concepts to dig deeper and deeper into the system layers. In addition, the user can modify and expand the system on any level by collecting new components out of the building blocks provided by the next level below it.

The user at this level now has an intuition of the overall layout of the city. He also has sub-maps of all the details and he knows how to read them. Depending on his personality, he may use these maps only when absolutely necessary, or he may use them to explore unknown territory. In contrast to the tourist, the Smalltalk user can even make modifications and new extensions to the city. The tools are there. The user decides if, when, and how he wants to use them.

Example of a System Description

The problem: Consider a small manufacturing company that has two



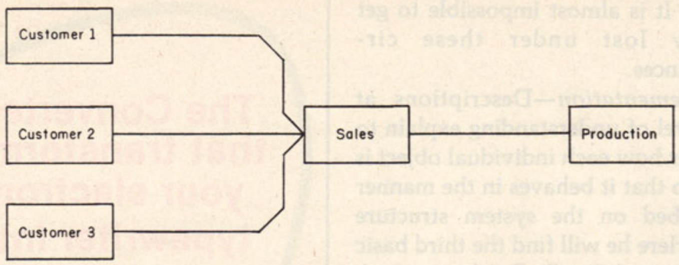


Figure 1: *A job-shop manufacturing company with its customers.*

departments: sales and production. The responsibility of the sales department is to find customers for any product the company can make, to contact the production department to find out when the product can be delivered, and to sign a contract with the customer. The responsibility of the production department is to manufacture each product as cheaply as possible at a specified level of quality and to have it finished on the promised date. When the production department has manufactured the product, it is dispatched to the customer through the sales department.

The system: A natural way to map this into a Smalltalk system would be to represent each department as an object. The function of the Sales object would be to keep track of the state of each sale in the following sequence:

1. Fill in and send proposals
2. Reserve the necessary resources in production for the product
3. Send contracts and packing notes to the customer

The function of the Production object would be to:

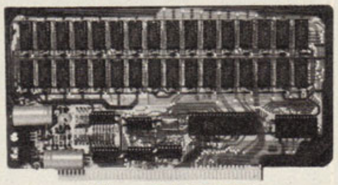
1. Keep track of commitments
2. Schedule the manufacture of products
3. Help keep the product quality
4. Control the manufacturing process to get the products completed on time

It also seems reasonable to include a third kind of object in our system: Customer objects. The purpose of these objects would be to act as a receptacle for the messages being passed from the company to the customer and from the customer to the company. The various objects with a set of reasonable communication channels is shown in figure 1.

The overall processing of an order: The Smalltalk system would be programmed to reflect everything of importance that takes place during the processing of an order and to support its user on every step. The process that takes place inside the Smalltalk system would, therefore, closely resemble the actual processing of an order. Let us assume the following real-life process, which is depicted in figure 2. A customer submits an intention to buy, a request for offer, to the company. The sales department books resources from the production department and returns an offer with the cost and delivery date to the customer. The customer answers with a purchase order. This is transcribed and passed from sales to production as a requisition. The product is manufactured in production, and a ready-note is sent to sales, which arranges for transport and sends packing notes to the customer.

In the Smalltalk system, the Sales object would help the user of the system in corresponding with the customer, in keeping track of progress, and in sending the required forms to the production department. The Production object would help the user in the planning and control of the manufacturing process.

In order to highlight the principles, we have made this a very simple system. The reader will have no difficulty in expanding it, for example, by adding an object for the accounting department that takes care of bill-



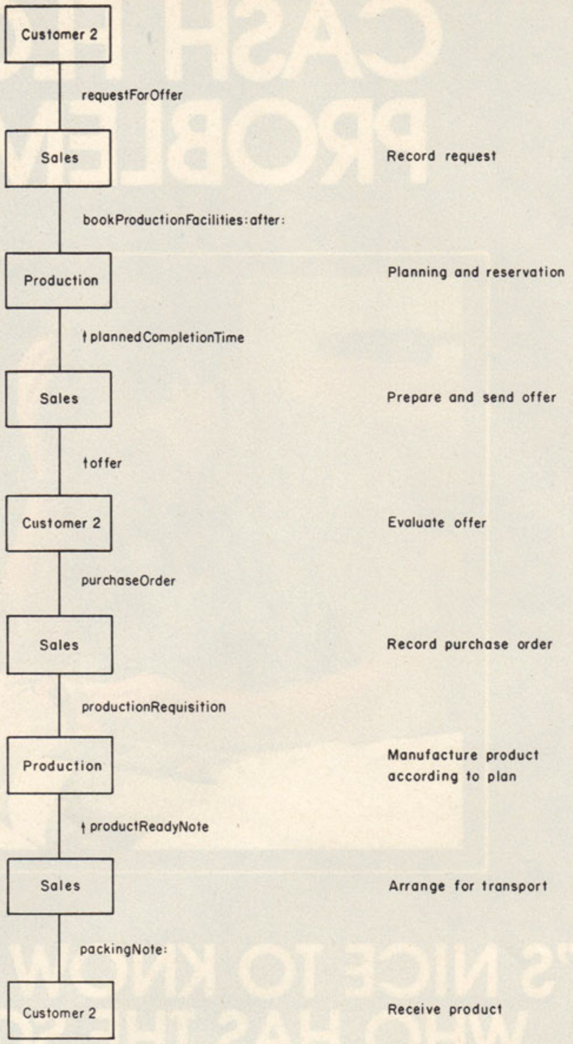


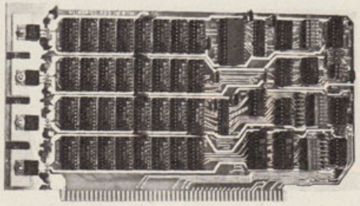
Figure 2: *The processing of an order. The Smalltalk system supports this processing through interaction with its owner in real-time.*

ing, an object for the warehouse that may or may not have the required product in stock, and so on. Also, figure 2 could probably be better documented on a Smalltalk computer by animating figure 1.

An Implementation Description

Let us inspect the Production object of figure 1 and see how it processes the message bookProductionFacilities: after:. When this message is received by the Production object, it consults its message dictionary to find the corresponding method. If the products were simple and the workshop small, the object could contain the current production plan directly and the method could go something like that shown in listing 1.

One of the instance variables of the Production object is the table productDuration which contains the time it takes to manufacture various products. Looking at this table, we find the duration for a product. In this simple example, there is only one resource, and we find the first available time slot for the product by sending self the message findFreePeriod: after:. This corresponds to calling a local subroutine in other systems. We then reserve the resource for our product in that period. (These two steps could have been combined into one, but the separation gives us more flexibility in varying the planning algorithm if we wish to do so later.)



Listing 1: *Smalltalk method for the message* bookProductionFacilities:after:.

```
bookProductionFacilities: productType after: earliestStartTime
    "Reserves production facilities for a new product of given type as soon as possible after the
        specified earliest starting time. Returns the planned completion time for the product."
  | duration plannedStartTime |
  duration ← productDuration at: productType.
  plannedStartTime ← self findFreePeriod: duration after: earliestStartTime.
  self reservePeriod: duration from: plannedStartTime.
  ↑ (plannedStartTime + duration)
```

Listing 2: *Alternate Smalltalk method for the message* bookProductionFacilities:after:.

```
bookProductionFacilities: productType after: earliestStartTime
    "Reserves production facilities for a new product of given type as soon as possible after the
        specified earliest starting time. Returns the planned completion time for the product."
  | productIdentification |
  productIdentification ← jobManager defineProduct: productType.
  jobManager schedule: productIdentification after: earliestStartTime.
  ↑ ( jobManager plannedCompletionTime: productIdentification).
```

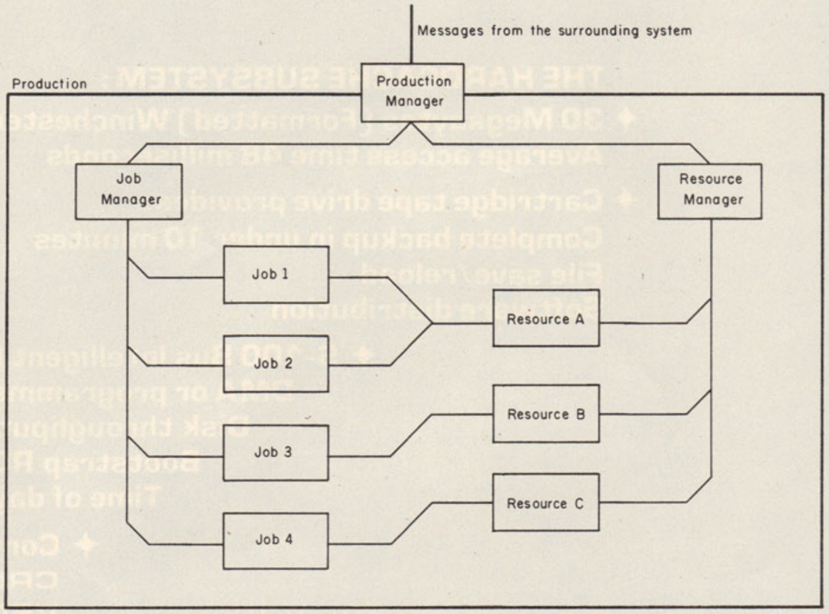


Figure 3: *The internals of the Production object.*

The planned completion time is returned to the sender, in this case the Sales object.

Lower-Level System Description

If the user wants more advanced aids for production control, the Production object would call upon the services of a subsystem of interconnected objects. A possible subsystem is shown in figure 3.

The entrance to the internals of the Production objects is through a Production Manager object; it is connected to a Job Manager object and a Resource Manager object.

The manufacturing of a product is split into a number of jobs. The available production facilities (people and machines) are split into a number of resources. Each job is to be performed by a single resource. A natural way to map this into a Smalltalk system is to represent each job by a Job object and each resource by a Resource object.

In this scheme, each Job object ensures that the job is performed by its resource within the available time. Similarly, each Resource object ensures that its resource is used in an efficient manner, that there is sufficient time available for preventive maintenance, and that there are no unacceptable overloads. The method in the Production object that handles the *bookProductionFacilities:after:* message could now be written as shown in listing 2.

One of the instance variables of the Production Manager object is a pointer to the Job Manager object. By using that pointer as a communication channel, the Production Manager object passes most of the work on to the Job Manager object. First, the Job Manager is asked to define the new product. The Job Manager creates the Job objects (see figure 3), links them to the proper Resource objects, and returns an identification that is to be used for future references to the product. The Job Manager is then asked to schedule the product for manufacturing as soon as possible after the given date. Finally, the Job Manager is asked when the product will be completed, and this value is returned to the outside world (in this case, to the Sales object). The planning process in the Production subsystem that is shown in figure 4 is controlled by this method.

Definition of New Objects

The first task of the Job Manager object is to define the new object. It receives message *defineProduct:* when this is to be done. The corresponding method could be something like that shown in listing 3. We are referencing two instance variables of the Job Manager object in this method: *productDescriptions* and *productionManager*. *productDescriptions* is an ordered collection with one member for each product type. Each of these members contains a sequence of small objects with the class, duration, and resource type for each of the jobs that go into the manufacture of such a product. *productionManager* contains a pointer to the Production Manager object. The result of the product creation is put into a third instance variable, the *productDic-*

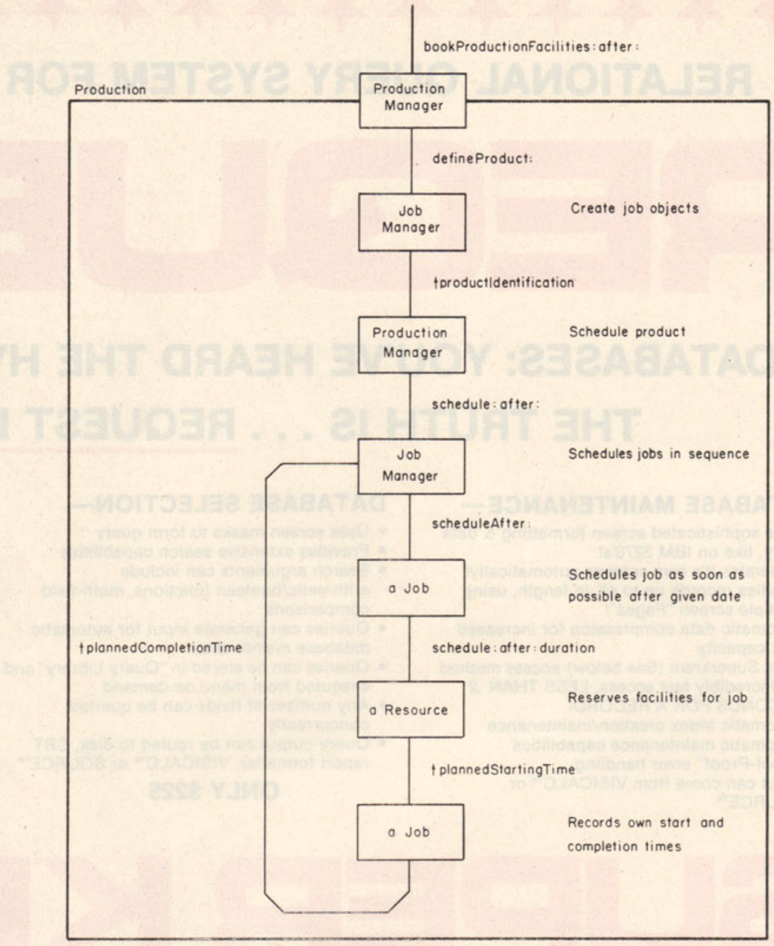


Figure 4: *A simple planning algorithm implemented in a Smalltalk system.*

Listing 3: *Smalltalk method for the message* defineProduct:.

```
defineProduct: productType
"To create a new product of given type. The corresponding Job objects are created and linked to
their resource objects."
|productIdentification jobDescriptions job jobList resourceObject
productIdentification ← self nextProductIdentification.
jobDescriptions ← productDescriptions at: productType.
jobList ← jobDescriptions collect:
    [ :description |
    job ← (description class) new.
    job duration: (description duration).
    resourceObject ← productionManager getResource: (description
    resourceType).
    job resource: resourceObject].
productDictionary at: productIdentification put: jobList.
↑ productIdentification.
```

tionary. In this dictionary, each key is a product identification; the corresponding entry is the sequence of job objects for that product.

The first line of code gets a new, unique identification for the new product. Next, the list of job descriptions is retrieved from the productSpecification collection. We then build the sequence of Job objects by going systematically through the job descriptions. For each description, we create a new Job object of the given class, feed it its duration, and let it

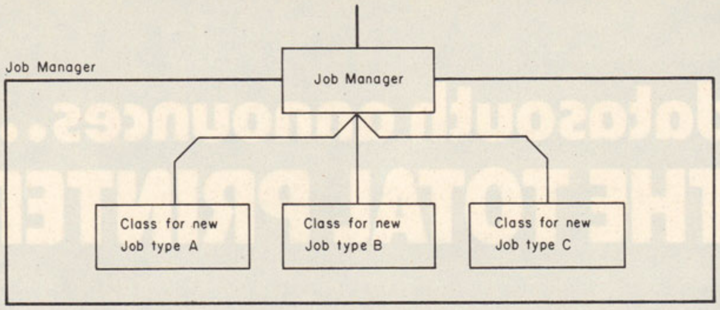


Figure 5: *Sub-objects in the Job Manager actually create the new Job objects.*

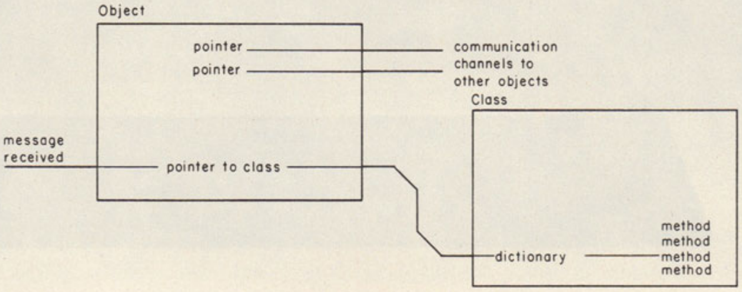


Figure 6: *All objects contain a pointer to a Class object that contains their message dictionary and methods.*

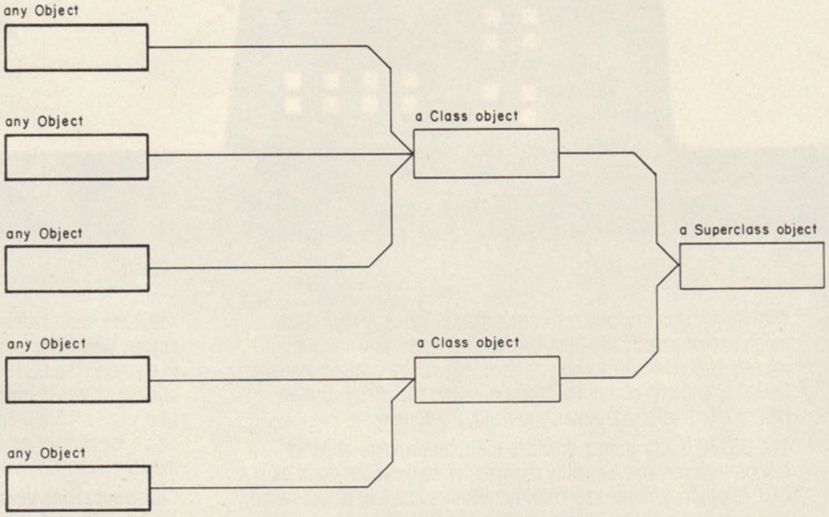


Figure 7: *The superclass-subclass chains of pointers. The user does not meet them unless he wants to become a real Smalltalk expert.*

connect itself to its Resource object. From figure 3, we see that there is no direct connection between the Job Manager object and the resources. We therefore have to go via the Production Manager object to get the pointer to the Resource object that we give to the new Job object.

We finally insert the new list of jobs into the productDictionary in the Production Manager object and return the product identification.

The Job Manager is built so that Job objects may belong to several different classes. The different Job objects created would all understand the same message protocols, but they would differ in their implementation. For example, a job might be: *wait for 24 hours while a resin glue is curing.* This does not need any resources, and the planning of such a job would be very simple—wait 24 hours. Another kind of job, such as pouring concrete, should not span a weekend, since joining old and new concrete could give weak spots in the product.

As is the case with Job objects, we often find that several objects share the same message protocols and process the messages with the same methods. Their only difference is that they appear in different places in the total system and that their instance variables point to different objects (their states are different). Such objects are created by the same class object and are said to belong to the same class.

The Smalltalk user should be able to "open up" the application object on the screen to see its component parts and to find out how they work together.

It would be very inefficient if each object of a class stored a replica of the message dictionary and all methods, and it would be extremely tiresome if we actually had to program each object by itself. We, therefore, use the concept of layering to let each and every object enlist the services of its class object in order to decode an incoming message and to select the proper method to process it. This mechanism is illustrated in figure 6. As in so many other parts of Smalltalk, we find a recursive argument.

Many classes are very similar; they differ only in the handling of a few messages. The different kinds of Job objects are a case in point. It seems reasonable to let a class object enlist the services of a superclass object whenever it is called upon to execute methods it shares with other classes.

Many classes will then share the same superclass; we get a tree-shaped class structure as shown in figure 7. Note that the purpose of this structure is convenience in programming and efficiency in implementation; it belongs on the lowest levels of the system hierarchy and is not part of the structure of the application system.

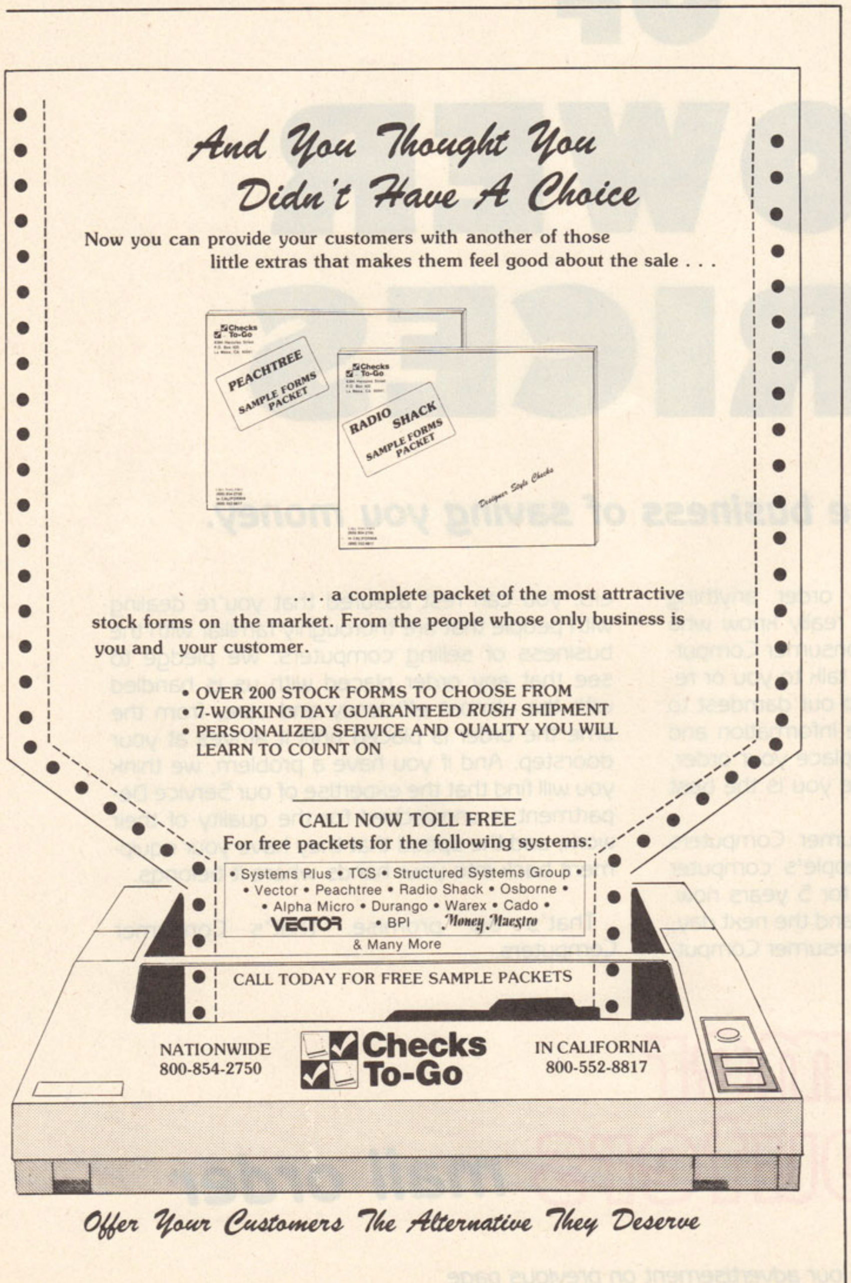
Future Experiments

When personal computing becomes sufficiently entertaining and interesting to become a widespread tool, the new user of a Smalltalk system is likely to begin by using its ready-made application systems for writing and illustrating documents, for designing aircraft wings, for doing homework, for searching through old court decisions, for composing music, or whatever. After a while, he may become curious as to how his system works. He should then be able to "open up" the application object on the screen to see its component parts and to find out how they work together. He could, for example, see something like figure 1 together with his usual user interface. By exercising the application commands, the computing process could be illustrated on the system diagram. Using Smalltalk to document itself in this manner should make it possible to make some novel and extremely powerful system description tools.

The next thing the user might want to do is to build new systems similar to the one he has been using. A *kit* of graphical building blocks would let the user compose a new system by editing the system diagram on the screen. While the Trip system (as described in reference 2) is not a proper kit, it could be a good source of ideas to the experimenter on building such systems.

Finally, the expert user would want to make his own kits. Even here, it is important that he sees only what he needs and that all unimportant details are suppressed. Since what is important in one context might be unimportant in another, and vice versa, the concepts of *filters* (see reference 1) will be an essential ingredient for the experimenter when he develops tools for these expert users.

Much experimenting needs to be done before we learn how to make systems that are self-documenting on any level and that provide a smooth and stumble-free transition from one level to the next. It is hoped that the availability of Smalltalk will lead to great activity in this field, to the benefit of all future computer users. ■



Checks To-Go
PEACHTREE SAMPLE FORMS PACKET
RADIO SHACK SAMPLE FORMS PACKET
Designer Style Checks

References

1. Goldberg, Adele and David Robson. "A Metaphor for User Interface Design." *Proceedings of the University of Hawaii Systems Science Symposium*, January 1979, Honolulu.
2. Gould, Laura and William Finzer. "A Study of TRIP: A Computer System for Animating Time-Rate-Distance Problems." *Proceedings of the IFIP World Conference on Computers in Education (WCCE-81)*, Lausanne, Switzerland, July 1981.
3. Ingalls, Daniel H H. "The Smalltalk-76 Programming System. Design and Implementation." *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

The Smalltalk Graphics Kernel

Daniel H H Ingalls
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Graphics are essential to the quality of an interactive programming system and to the interactive applications that go along with such a system. Qualitatively, people think with images, and any system that is incapable of manipulating images is incapable of augmenting such thought. Quantitatively, a person can visually absorb information equivalent to millions of characters a second, while the normal rate for reading text is less than 100 characters a second.

For the graphical interaction cycle to be complete, a computer system must provide a channel for input in the visual domain as well. While the projection of images from the realm of thought into the space of electronic information seems an impossible task, a well-designed pointing device can effectively harness the computer's graphical output capability to express graphical input from the user. Given such a pointing device, the process of selecting from graphical objects, such as text displayed on the screen, is natural and rapid. By tracking the pointer with a program that simulates a pen or paintbrush, the visual input channel can be extended to include line drawing and freehand sketches.

The purpose of graphics in the Smalltalk system is to support the *reactive principle*:

> *Any object accessible to the user should be able to present itself in a meaningful way for observation and manipulation.*

Meaningful presentation of any object in the system demands maximum control over the display medium, and many technologies fall short in this respect. One approach that provides the necessary flexibility is to allow the brightness of every discernible point in the displayed image to be independently controlled. The simplest implementation of this approach is a contiguous block of storage in which the setting of each *bit* (1 or 0) is *mapped* into dark or light illumination of the corresponding picture element, or *pixel*, when displaying or combining with other images. The block of storage is thus referred to as a *bitmap*, and this type of display is called a *bitmap display*. The simplest form of bitmap allows only two brightness levels, white and black. The Smalltalk-80 graphics system is built around this model.

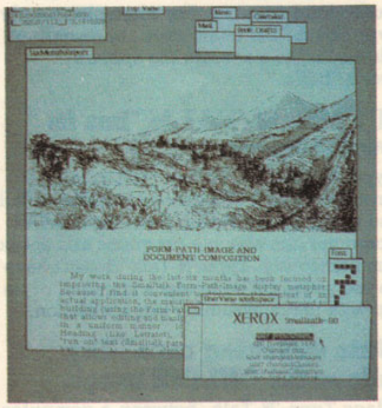


Photo 1: *An example of a Smalltalk-80 video display. Note the multiple windows, the combinations of text and graphics, and the pointer in the window marked "UserView workspace."*

Photo 1 shows a typical view of the Smalltalk-80 system, and it illustrates the wide range of graphical idiom implied by the reactive principle. Rectangular areas of arbitrary size are filled with white, black, and various halftone patterns. Text, in various typefaces, is placed on the screen from stored images of the individual characters. Halftone shades are "brushed" by the user to create freehand paintings. Moreover, although not shown on the printed page, images on the display may be moved or sequenced in time to provide animation.

Graphical Storage—Forms

Simple images are represented by instances of class Form. A Form has height and width and a bitmap that indicates the white and black regions of the particular image being represented. Consider, for example, the arrow-shaped Form that appears in the lower-right window of the screen image in photo 1. The internal representation of this Form is depicted in figure 1. Its height is 16, its width is 8, and its appearance is described by the pattern of ones and zeros (shown as light and dark squares) in its bitmap. The height and width of the Form serve to impose the

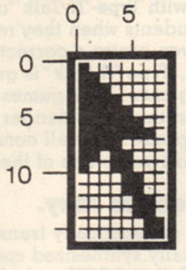


Figure 1: *A simple* Form *representing the cursor in photo 1.*

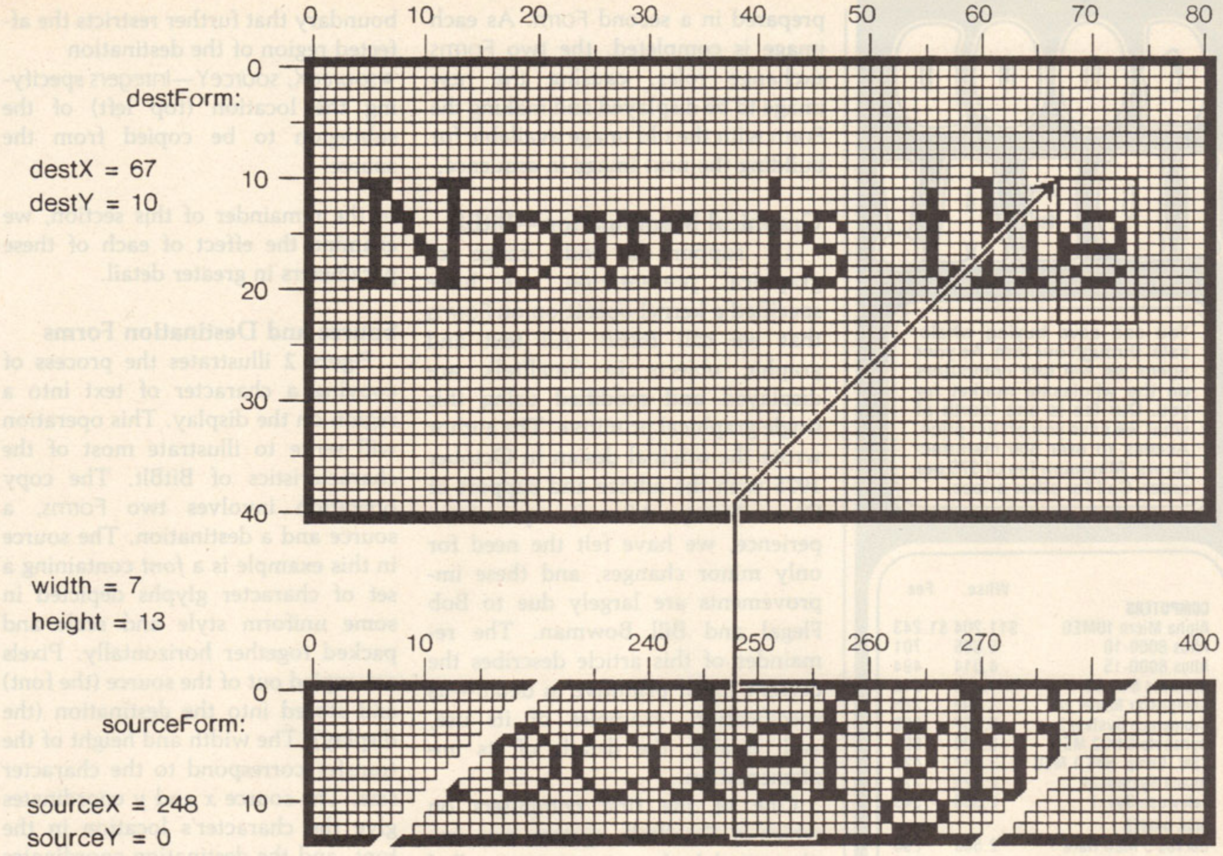


Figure 2: *Copying a character of text from a source* Form *(bottom) to a destination* Form *(top).*

appropriate two-dimensional ordering on the otherwise unstructured data in the bitmap. We will return to the representation of Forms in more detail later in this article.

A complex image can be represented in either of two ways: by a very large Form, or by a structure that includes many Forms and rules for combining and repeating them in order to produce the desired image. The freehand drawing in the center of photo 1 is an example of the former, and the text below it is an example of the latter.

The large unstructured Form has an additional use of great importance: it can be presented to the display hardware as a buffer in memory of the actual data to be shown on the display terminal. We refer to the Form which is so used as the displayForm. Since the interface to the hardware is through a Form, there is no difference between combining images internally and displaying them on the screen. Animation can be done simply in this manner: one Form serves as the displayForm while the next image to be displayed is

prepared in a second Form. As each image is completed, the two Forms exchange roles, causing the new image to be displayed and making the Form with the old image available for building the next image in sequence.

Graphical Manipulation—BitBlt

To support a wide range of graphical presentation, we have specified a kernel operation on Forms that we call *BitBlt*. All text and graphic objects in Smalltalk are displayed and modified using this single graphical primitive. The author wrote the original design in October 1975 with the advice and support of Diana Merry. After five years' experience, we have felt the need for only minor changes, and these improvements are largely due to Bob Flegal and Bill Bowman. The remainder of this article describes the current BitBlt primitive in detail—its specification, examples of its use, and, finally, the details of its implementation.

One of the first computers on which a Smalltalk system was implemented had an instruction called BLT for *block transfer* of 16-bit words. The name BitBlt derives from the generalization of data transfer to arbitrary bit locations, or pixels. BitBlt is intentionally a very general operation, although most applications of it are graphically simple, such as "move this rectangle of pixels from here to there."

A specific application of BitBlt is governed by a list of parameters that includes:

- destForm—a Form into which pixels will be stored by BitBlt
- sourceForm—a Form from which pixels may be copied
- halftoneForm—a Form containing a spatial halftone pattern
- combinationRule—an Integer specifying the rule for combining corresponding pixels of the sourceForm and destForm
- destX, destY, width, height—Integers specifying the rectangular subregion to be filled in the destination
- clipX, clipY, clipWidth, clipHeight—Integers specifying a rectangular boundary that further restricts the affected region of the destination
- sourceX, sourceY—Integers specifying the location (top left) of the subregion to be copied from the source

In the remainder of this section, we examine the effect of each of these parameters in greater detail.

Source and Destination Forms

Figure 2 illustrates the process of copying a character of text into a region on the display. This operation will serve to illustrate most of the characteristics of BitBlt. The copy operation involves two Forms, a source and a destination. The source in this example is a *font* containing a set of character glyphs depicted in some uniform style and scale and packed together horizontally. Pixels are copied out of the source (the font) and stored into the destination (the display). The width and height of the transfer correspond to the character size. The source *x* and *y* coordinates give the character's location in the font, and the destination coordinates specify the position on the display where its copy will appear.

Clipping Rectangle

In its specification, BitBlt includes a rectangle that limits the region of the destination that can be affected by its operation, independent of the other destination parameters. We call this rectangle the *clipping rectangle*. Often it is desirable to display a partial *window* onto larger scenes, and the clipping rectangle ensures that all picture elements fall inside the bounds of the window. By its inclusion in the BitBlt primitive, the clipping function can be done efficiently and in one place, rather than being replicated in all application programs. Figure 3 illustrates the result of imposing a clipping rectangle on the example of figure 2. Pixels that would have been placed outside the clipping rectangle (the left edge of the "N" and half of the word "the") have not been transferred. If other characters had fallen above or below this rectangle, they would have been clipped similarly.

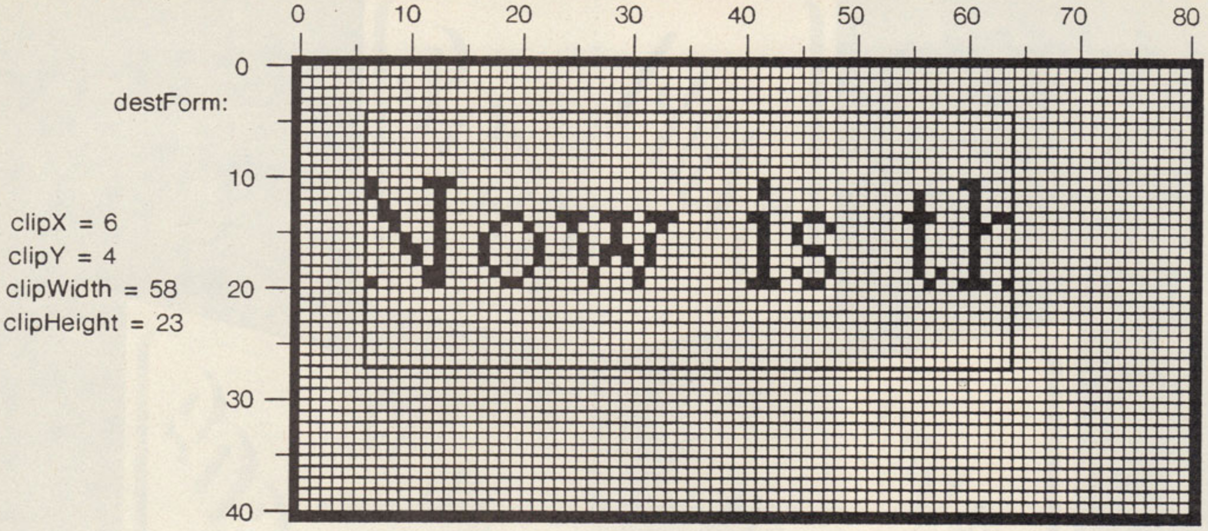


Figure 3: *An example of using a clipping window on the illustration in figure 2.*

Halftone Form

It is often desirable to fill areas with a regular pattern that gives the effect of gray shading or texture. To this end, BitBlt provides for reference to a third Form (halftoneForm) containing the desired pattern. This Form is restricted to a height and width of 16. When halftoning is specified, this pattern is effectively repeated every 16 units horizontally and vertically over the entire destination. There are four "modes" of supplying pixels from the source and halftone controlled by eliding (supplying nil for) sourceForm or halftoneForm:

•*Mode 0*—No source, no halftone (supplies solid black)

•*Mode 1*—Halftone only (supplies halftone pattern)

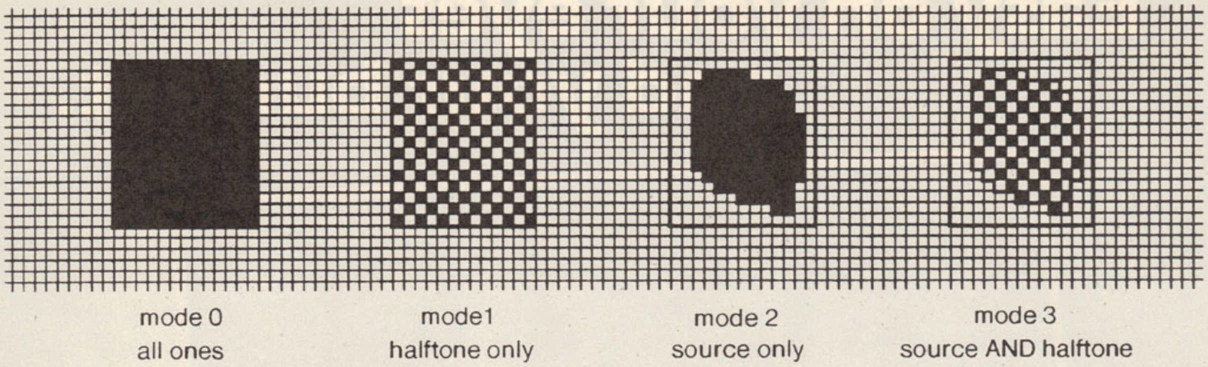


Figure 4: *BitBlt's four possible source modes.*

- *Mode 2*—Source only (supplies source pixels)
- *Mode 3*—Source AND halftone (supplies source bits masked by halftone pattern)

Figure 4 illustrates the effect of these four modes with the same source and destination and a regular gray halftone.

Combination Rule

The examples above have all stored their results directly into the destination. There are actually many possible rules for combining each source element S with the corresponding destination element D to produce the new destination element D′. Such a rule must specify a white or black result for each of the four cases of source being white or black and destination being white or black.

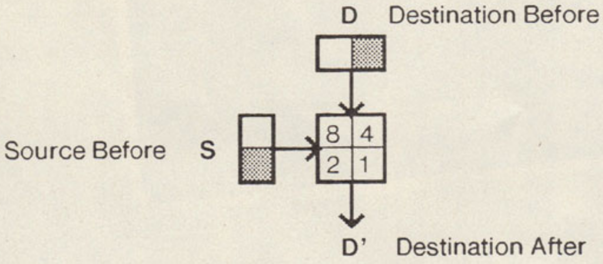


Figure 5: *A BitBlt combination diagram. This diagram, when filled in, specifies the effects of a given combination (or "rule") on all combinations of dark and light source and destination cells. Each combination is given a number equal to the sum of the cells that are darkened. See figure 6 for examples.*

Figure 5 shows a box with four cells corresponding to the four cases encountered when combining source (S) and destination (D). For instance, the cell numbered 2 corresponds to the case where the source was black and the destination was white. By appropriately filling the four cells with white or black, the box can be made to depict any combination rule (there are sixteen possible rules altogether). The numbers in the four cells relate the rule as depicted to the integer value that selects that rule. For instance, to specify that the result

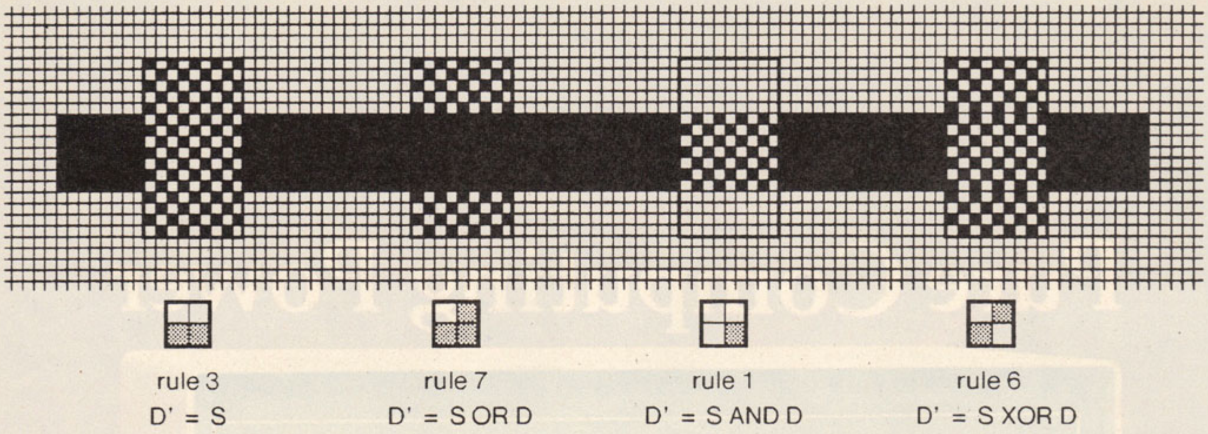


Figure 6: *Four common combination rules.*

should be black wherever the source or destination (or both) was black, we would blacken the cells numbered 4, 2, and 1. The associated integer for specifying that rule is the sum of the blackened cell numbers, or 4 + 2 + 1 = 7.

Figure 6 illustrates four common combination rules graphically. Each is described by a combination diagram, its integer rule number, and the actual logical function being applied. The earlier case of ORing can be seen in left center of the figure. This case is often described as painting "under" the destination because existing black areas remain black.

Smalltalk Access to BitBlt

In this section, we present the Smalltalk interface to BitBlt and take a detailed look at the application of BitBlt to text display and line drawing. In preparation, you will need some additional context, which we present here before describing class BitBlt.

Besides class Form, two additional classes are used extensively in working with stored images, Point and Rectangle. Points contain x and y coordinate values and are used for referring to pixel locations relative to the top left corner of a Form (or other point of reference). By convention, x increases to the right and y down, consistent with the layout of text on a page and the direction of TV scanning. A Rectangle contains two Points: origin, which specifies the top left corner, and corner, which indicates the bottom right corner of the region described. Class Point provides protocol for access to the coordinates and for various useful operations such as translation and scaling. Class Rectangle provides protocol for access to all the coordinates involved and other operations such as intersection with other rectangles. It may be useful to note the parallel between classes Point, Rectangle, Form and classes Number, Interval, IndexedCollection. Numbers index Collections and Points index Forms. Intervals select subCollections, and Rectangles select subForms.

Figure 7 shows the complete representation of the Form shown in figure 1. The width and height are stored as Integers. The actual pixels are stored in a separate instance of class Bitmap. Bitmaps have almost no protocol, since their sole purpose is to provide storage for Forms. They also have no intrinsic dimensionality, apart from that projected by their own Form, although the figure retains this structure for clarity. It can be seen that space has been provided in the Bitmap for a width of 16; this is a manifestation of the hardware organization of storage and processing into 16-bit *words*. Bitmaps are allocated with an integral number of words for each row of pixels. The integral constraint on row size facilitates movement from one row to the next during the operation of BitBlt and during scanning of the display screen by the hardware. While this division of memory into words is significant at the primitive level, it is encapsulated in such a way that none of the higher-level graphical components in the system need consider word size.

Class BitBlt

The most basic interface to BitBlt is through a class of the same name. Each instance of BitBlt contains the parameters necessary to specify a BitBlt operation. The BitBlt protocol includes messages for initializing the parameters and one message, copyBits, that causes the primitive operation to take place. The class template for BitBlt is given in table 1.

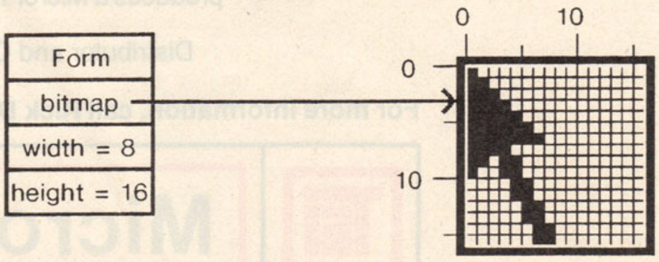


Figure 7: *The complete representation of figure 1.*

class name	BitBlt
superclass	Object
instance variable names	destForm sourceForm halftoneForm combinationRule destX destY width height clipX clipY clipWidth clipHeight sourceX sourceY
instance messages and methods	

```
setup

destForm: form1 sourceForm: form2 halftoneForm: form3 rule:
   rule destRectangle: destRectangle clipRectangle: clipRectangle
   sourceOrigin: sourceOrigin | |

    destForm ← form1.
    sourceForm ← form2.
    halftoneForm ← form3.
    combinationRule ← rule.

    destX ← destRectangle minX.
    destY ← destRectangle minY.
    width ← destRectangle width.
    height ← destRectangle height.

    clipX ← clipRectangle minX.
    clipY ← clipRectangle minY.
    clipWidth ← clipRectangle width.
    clipHeight ← clipRectangle height.

    sourceForm = = nil ifFalse:
       [sourceX ← sourceOrigin x.
       sourceY ← sourceOrigin y].

    self copyBits

operations

copyBits | | < primitive >
```

Table 1: *Class template for class* BitBlt.

The state held in an instance of BitBlt allows multiple operations in a related context to be performed without the need to repeat all the setup. For example, when displaying a scene in a display window, the destination Form and clipping rectangle will not change from one operation to the next. This situation occurs frequently in the graphics kernel, as demonstrated in the following section.

Image Synthesis of Text

Much of the graphics in the Smalltalk system consists of text and lines. These high-level entities are synthesized by repeated invocation of BitBlt. In this section and the next, we examine these two important applications more closely.

One of the advantages derived from BitBlt is the ability to store fonts compactly and to display them using various combination rules. The compact storage arises from the possibility of packing characters horizontally one next to another (as shown in figure 2), since BitBlt can extract the relevant bits if supplied with a table of left x coordinates of all the characters. This is called a *strike* format, from the typographical term meaning a contiguous display of all the characters in a font.

The scanning and display of text is performed in the Smalltalk-80 system by a subclass of BitBlt. This subclass inherits all the normal state, with destForm indicating the Form in which text is to be displayed and sourceForm indicating a Form containing all the character glyphs side by side (as in figure 2). In addition, this subclass defines further state information, including:

- text—a String of Characters to be displayed
- textPos—an Integer giving the current position in text

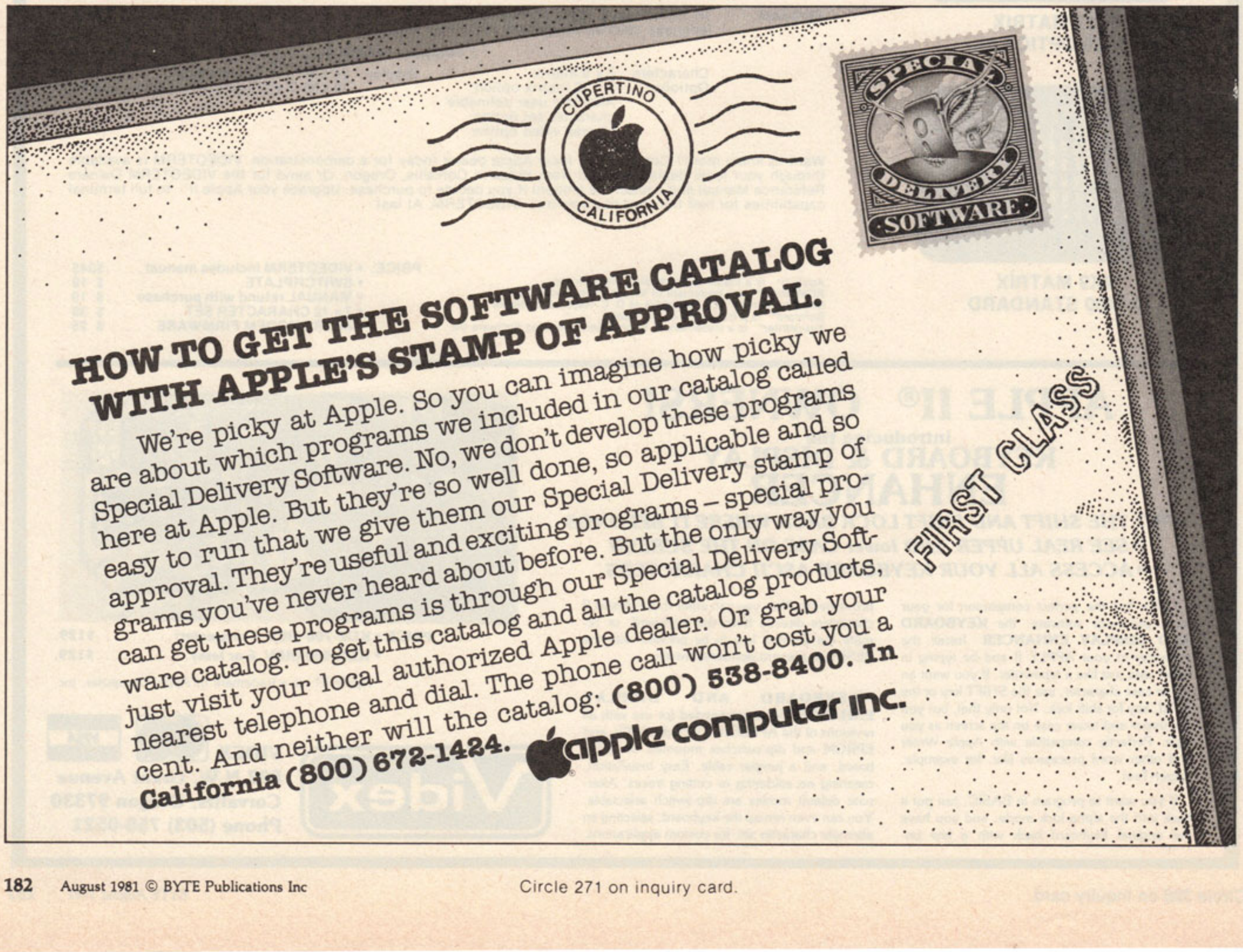
Listing 1: *The* scanWord: *method scans or prints text.*

```
scanWord: endRun
  | charIndex |
  < primitive >   "May be implemented internally for speed"
  [charIndex > endRun] whileTrue:
     [charIndex ← text at: textPos.                         "pick character"
     (exceptions at: charIndex) > 0                         "check exceptions"
        ifTrue: [↑ exceptions at: charIndex].
     sourceX ← xTable at: charIndex.                        "left x of character in font"
     width ← (xTable at: charIndex + 1) − sourceX.          "up to left of next char"
     printing ifTrue: [self copyBits].                      "print the character"
     destX ← destX + width.                        "advance by width of character"
     destX > stopX ifTrue: [↑ stopXCode].                   "passed right boundary"
     textPos ← textPos + 1].                                "advance to next character"
  textPos ← textPos − 1.
  ↑ endRunCode
```

- xTable—an Array of Integers giving the left *x* location of each character in sourceForm
- stopX—an Integer that sets a right boundary past which the inner loop should stop scanning
- exceptions—an Array of Integers that, if non-zero, indicate that the corresponding characters must be specially handled

Once an instance has been initialized with a given font and text location, the scanWord: loop given in listing 1 will scan or print text until some horizontal position (stopX) is passed, a special character (determined from exceptions) is found, or the end of this range of text (endRun) is reached.

The check on exceptions handles many possibilities in one operation. The space character may have to be handled exceptionally in the case of text that is padded to achieve a flush right margin. Tabs usually require a computation or table check to determine their width. Carriage return is also identified in the check for exceptions. Character codes beyond the range given in the font are detected similarly and are usually handled by showing an exceptional character, such as a little lightning bolt, so that they can be seen and corrected. The printing flag can be set false to allow the same code to *measure* a line (break at a word boundary) or to find where the cursor points. While this provision may seem over-general, two benefits (besides compactness) are derived from that generality. First, if one makes a change to the basic scanning algorithm, the parallel functions of measuring, printing, and cursor tracking are sure to be synchronized. Second, if a primitive implementation is provided for the loop, it exerts a threefold leverage on the system performance. The scanWord: loop is designed to be amenable to such primitive implementation; that is, the interpreter may intercept it and execute primitive code instead of the Smalltalk code shown. In this way, much of the setup overhead for copyBits can be avoided at each character, and an entire word or more can be displayed



Listing 2: *The* drawLoopX:Y: *method draws lines.*

```
drawLoopX: xDelta Y: yDelta
  | dx dy px py p i |
  < primitive >
  dx ← xDelta sign.
  dy ← yDelta sign.
  px ← yDelta abs.
  py ← xDelta abs.
  self copyBits.                  "first point"

  py > px
    ifTrue:                       "more horizontal"
      [p ← py/ /2.
       1 to: py do:
        [:i | destx ← destx + dx.
        (p ← p − px) < 0 ifTrue: [desty ← desty + dy. p ← p + py].
        self copyBits]]
    ifFalse:                      "more vertical"
      [p ← px/ /2.
       1 to: px do:
        [:i | desty ← desty + dy.
        (p ← p − py)<0 ifTrue: [destx ← destx + dx. p ← p + px].
        self copyBits]]
```

Listing 3: *Methods for image magnification. @ is a shorthand message that returns a new* Point *whose x-value is the receiver (on the left) and whose y-value is the argument (on the right).* Points *respond to the + and * messages by distributing them over each of the coordinates.*

```
magnify: rect by: scale spacing: spacing
  | wideForm bigForm |                                  "First expand horizontally"
  wideForm ← Form extent: (rect width * scale x) @ rect height.
  wideForm spread: rect from: self by: scale x
    spacing: spacing x direction: 1 @ 0.
  bigForm ← Form extent: rect extent * scale.          "Then expand vertically"
  bigForm spread: wideForm asRectangle from: wideForm by: scale y
    spacing: spacing y direction: 0 @ 1.
  ↑ bigForm

spread: rect from: sourceForm by: scale spacing: spacing
 direction: dir
  | slice sourcePt |
  slice ← Rectangle origin: 0 @ 0 extent: dir transpose * self extent + dir.
  sourcePt ← rect origin.     "transpose returns a Point with swapped coordinates"
  1 to: (rect extent dot: dir) do:     "dot product selects direction of stretch"
    [:i |     "slice up the original image"
    self copy: slice from: sourcePt in: sourceForm rule: STORing.
    sourcePt ← sourcePt + dir. slice moveby: dir * scale].
  1 to: scale − spacing − 1 do:
    [:i |     "smear out the slices, leave some space"
    self copyAllTo: 1 @ 0 in: self rule: ORing]
```

directly. Conversely, the Smalltalk text and graphics system requires implementation of only the one primitive operation to provide full functionality.

Line Drawings, Image Synthesis

The same design principle applies in the support for drawing lines. By using BitBlt, one algorithm can draw lines of varying widths, different halftone "color," and any combination rule. To draw a line, an instance of BitBlt is initialized with the appropriate destination Form and clipping window, and with a source that can be any Form to be applied as a pen shape along the line. Starting from the stored destX and destY, the line-drawing loop, drawLoopX:Y: (listing 2), accepts *x* and *y* delta values and *x* and *y* step values as necessary, calling copyBits at each point along the line. The method used is the Bresenham plotting algorithm (*IBM Systems Journal*, Volume 4, Number 1, 1965). It chooses a principal direction and maintains a variable, p. When p's sign changes, it is time to move in the minor direction as well. This procedure is another natural unit to be implemented as a primitive, since the computation is trivial and the setup in copyBits is almost all constant from one invocation to the next.

Image Processing

We have seen how BitBlt can copy shapes and, in the foregoing examples, how repeated invocation can synthesize more complex images such as text and lines. BitBlt is also useful in the manipulation of existing images. For example, text can be made to look bold by ORing over itself, shifted right by one pixel. Just as complex images can be built from simple ones, complex processing can be achieved by repeated application of simple operations. Here, we present three examples of such structural manipulation: magnification, rotation, and the game of Life. These examples were devised by the author in collaboration with Ted Kaehler.

As we shall see in the next two sections, many applications of BitBlt are very simple, such as filling a Form with white, or copying all of one Form to some location in another. Smalltalk provides for such casual use of BitBlt through a wide range of simple messages to class Form, such as:

```
someForm fillAll: white.
someForm copyAllTo:
    destLocation in: destForm.
```

We will not list all such messages here. In the examples that follow, the reader should be able to infer the meaning from the message names and the accompanying explanations.

Magnification

It is often useful to magnify an image for closer scrutiny and especially to allow convenient alteration of stored Forms. Photo 1 shows this function providing user control over the font used for display of text.

Listing 4: *The* rotate *method. This method rotates an image of size 2^n by 2^n one quarter-turn clockwise.*

```
rotate | mask temp quad |
   temp ← Form extent: self extent.
   mask ← Form extent: self extent.        "set up the first mask"
   mask copy: mask asRectangle halftone: white rule: STORing.
   mask copy: mask asRectangle/2 halftone: black rule: STORing.
   quad ← self width/2. "the size of a quadrant"
   [quad > = 1] whileTrueDo:
      [  "First exchange left and right halves"
      mask copyAllTo: 0 @ 0 in: temp rule: STORing.
      mask copyAllTo: 0 @ quad in: temp rule: ORing.
      self copyAllTo: 0 @ 0 in: temp rule: ANDing.
      temp copyAllTo: 0 @ 0 in: self rule: XORing.
      temp copyAllFrom: quad @ 0 in: self rule: XORing.
      self copyAllTo: (0 − quad) @ 0 in: self rule: ORing.
      temp copyAllTo: quad @ 0 in: self rule: XORing.
         "Then flip the diagonals"
      self copyAllTo: 0 @ 0 in: temp rule: STORing.
      temp copyAllFrom: quad @ quad in: self rule XORing.
      mask copyAllTo: 0 @ 0 in: temp rule: ANDing.
      temp copyAllTo: 0 @ 0 in: self rule: XORing.
      temp copyAllTo: quad @ quad in: self rule: XORing.
         "Compute the next fine mask"
      mask copyAllFrom: (quad/2) @ (quad/2) in: mask rule: ANDing.
      mask copyAllTo: quad @ 0 in: mask rule: ORing.
      mask copyAllTo: 0 @ quad in: mask rule: ORing.
      quad ← quad/2]
```

The character for "7" has been presented magnified nine times. Using a pointing device, the user has blackened some cells to provide a European style "7," and the result can be seen in both the upper-left and lower-right windows on the screen.

A simple way to magnify a stored Form would be to copy it to a larger Form, making a big dot for every little dot in the original. For a height h and width w, this would take $h \times w$ operations. The algorithm presented in listing 3 (as two messages to class Form) uses only a few more than $h + w$ operations.

The magnification proceeds in two steps. First, it slices up the image into vertical strips in wideForm separated by a space equal to the magnification factor. These are then smeared, using the ORing function, over the intervening area to achieve the horizontal magnification. The process is then repeated from wideForm into bigForm, with horizontal slices separated and smeared in the vertical direction, achieving the desired magnification. Figure 8 illustrates the progress of the above algorithm in producing the magnified "7" shown in photo 1.

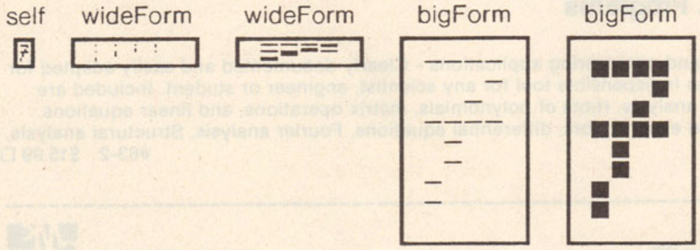


Figure 8: *Magnification with BitBlt. See the text for more details.*

Rotation

Another useful operation on images is rotation by a multiple of 90 degrees. Rotation is often thought to be a fundamentally different operation from translation, and this point of view would dismiss the possibility of using BitBlt to rotate an image. However, the reader must consent that the first transformation shown in figure 9 is a step toward rotating the image shown: all that remains is to rotate the insides of the four cells that have been permuted. The remainder of the figure shows each of these cells being further subdivided, its cells being similarly permuted, and so on. Eventually each cell being considered contains only a single pixel. At this point, no further subdivision is required, and the image has been faithfully rotated!

Each transformation shown in figure 9 would appear to require successively greater amounts of computation, with the last one requiring several times more than $h \times w$ operations. The tricky aspect of the algorithm below is to permute the subparts of every subdivided cell at once, thus performing the entire rotation in a constant times $log_2(h)$ operations. The parallel permutation of many cells is accomplished with the aid of two auxiliary Forms. The first, mask, carries a mask that selects the upper left quadrant of every cell; the second, temp, is used for temporary storage. A series of BitBlt operations exchanges the right and left halves of every cell, and then another series ex-

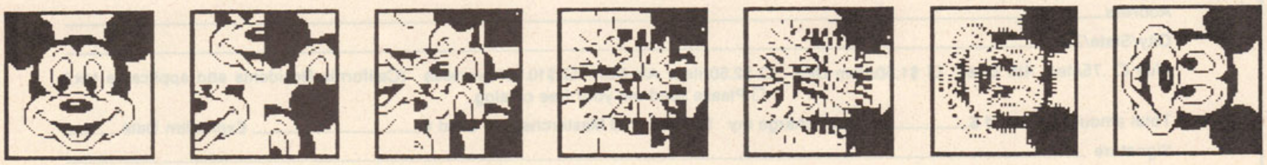


Figure 9: *Image rotation with BitBlt. See the text for more details.*

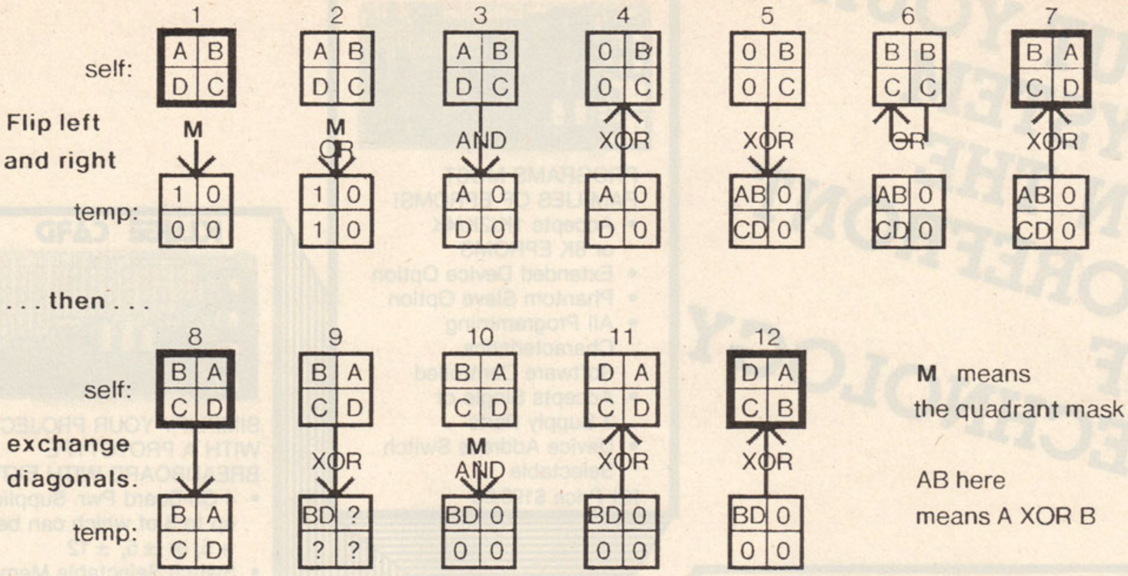


Figure 10: *Permuting four quadrants of a cell.*

changes the diagonal quadrants, achieving the desired permutation. The complete method for rotation is given in listing 4.

Figure 10 traces the state of temp and self after successive operations. The offsets of each operation are not shown, though they are given in the prcʒram listing. After twelve operations, the desired permutation has been achieved. At this point, the mask evolves to a finer grain, and the process is repeated for more, smaller cells. Figure 11 shows the evolution of the mask from the first to the second stage of refinement. The reader will note that the algorithm presented here for rotation is applicable only to square forms whose size is a power of two. The extension of this technique to arbitrary rectangles is more involved and is left as an exercise for the reader. A somewhat simpler exercise is to apply the above technique to horizontal and vertical reflections about the center of a rectangle.

The Game of Life

John Conway's game of Life is probably well known to readers of BYTE. It is a fairly simple rule for successive populations of a bitmap. The rule involves the neighbor count for each cell—how many of the eight adjacent cells are occupied? Each cell will be occupied in the next generation if it has exactly three neighbors, or if it was occupied and has exactly two neighbors. This is explained as follows: three neighboring organisms can give birth in an empty cell, and an existing organism will die of exposure with less than two neighbors or from overpopulation with more than three neighbors. Since BitBlt cannot add, it would seem to be of no use in this application. However, BitBlt's combination rules do include the rules for partial sum (XOR) and carry (AND). With some ingenuity and a fair amount of extra storage, the next generation of any size of bitmap can be computed using a *constant* number of BitBlt operations.

Listing 5 gives the method for nextLifeGeneration. As shown in figure 12, the number of neighbors is represented using three image planes for the 1s bit, 2s bit, and 4s bit of the neighbor count in binary. The 8s bit can be ignored, since there are no survivors in that case, which is equivalent to zero (the result of ignoring the 8s bit). This Smalltalk method is somewhat wasteful, as it performs the full carry propagation for each new neighbor, even though nothing will propagate into the 4-plane until at least the fourth neighbor. Some readers may enjoy improving upon this algorithm.

Many other image-processing tasks can be performed with BitBlt. The author has built a complete optical character-recognition system for Sanskrit text using the various combination rules and an operation that counts the number of black bits in any rectangle (how would you do it?).

Bitmap processing is ideally suited to VLSI (very large scale integration) implementation. Readers who are interested in this direction should check the proceedings of the Design Automation Conference, June 1981, for "Parallel Bitmap Processor," by Tom Blank, Mark Stefik, and Willem vanCleemput.

Efficiency Considerations

Our original specification for BitBlt has been published elsewhere

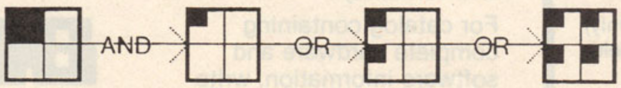


Figure 11: *Refinement of the quadrant mask.*

Listing 5: *The* nextLifeGeneration *method. This method calculates the next Life generation given the BitBlt bitmap of the current generation. See figure 12.*

```
nextLifeGeneration | nbr1 nbr2 nbr4 carry2 carry4 |
   nbr1 ← Form new extent: self extent + (2 @ 2).          "temp areas larger by 1"
   nbr2 ← Form new extent: self extent + (2 @ 2).          "bit all around"
   nbr4 ← Form new extent: self extent + (2 @ 2).
   carry2 ← Form new extent: self extent + (2 @ 2).
   carry4 ← Form new extent: self extent + (2 @ 2).
   (1 @ 1) eightNeighbors do:
      [:delta|     "delta equals a different neighbor-offset each time through this loop"
      carry2 copyAllFrom: 0 @ 0 in: nbr1 rule: STORing.
      carry2 copyAllFrom: delta in: self rule: ANDing.       "carry into 2"
      nbr1 copyAllFrom: delta in: self rule: XORing.         "sum 1"
      nbr2 copyAllTo: 0 @ 0 in: carry4 rule: STORing.
      carry2 copyAllTo: 0 @ 0 in: carry4 rule: ANDing.       "carry into 4"
      carry2 copyAllTo: 0 @ 0 in: nbr2 rule: XORing.         "sum 2"
      carry4 copyAllTo: 0 @ 0 in: nbr4 rule: XORing].        "sum 4"
   nbr2 copyAllTo: 1 @ 1 in: self rule: ANDing.  "perform logic to determine the survivors"
   nbr2 copyAllTo: 0 @ 0 in: nbr1 rule: ANDing.     "(2s AND self) OR (2s AND 1s))"
   nbr1 copyAllTo: 1 @ 1 in: self rule: ORing.      "...all AND (NOT 4s)"
   nbr4 copyAllTo: 0 @ 0 in: self rule: NOTANDing       "store next generation"
                                                         "over self"
```

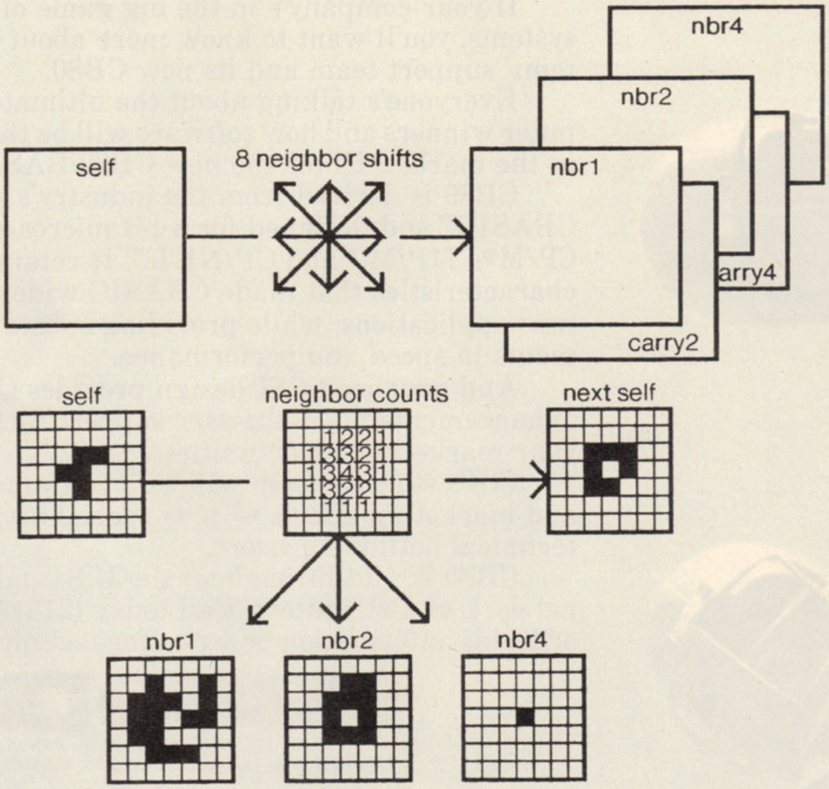
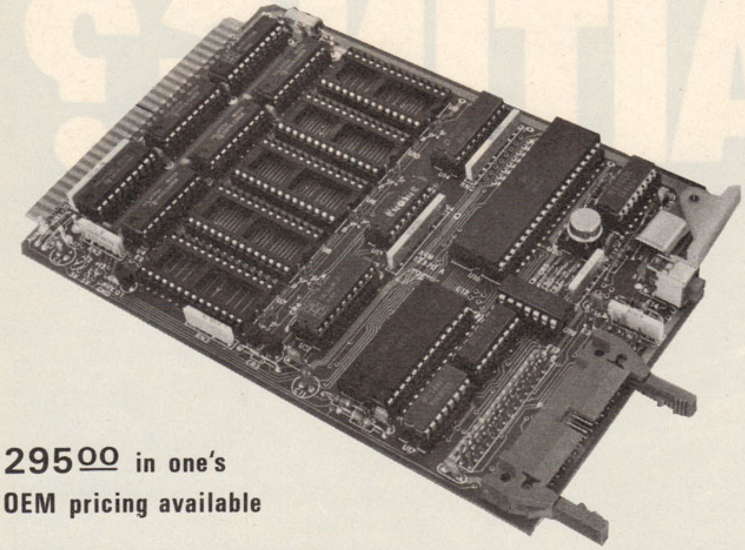


Figure 12: *Counting neighbors in the game of Life.*

(Newman and Sproull, *Principles of Interactive Computer Graphics*, 2nd edition, McGraw-Hill, 1979) under the name *RasterOp*. The implementation described in that reference can easily be extended to include the full set of combinations, and the addition of clipping is also straightforward. Here, we add a few notes on efficiency gathered from experience.

BitBlt is so central to the user interface that any improvement in its performance has considerable effect on the interactive quality of the system

Circle 135 on inquiry card. ⟶



as a whole. In normal use of the Smalltalk-80 system, most calls on BitBlt are either in the extreme microscopic or macroscopic range. Let us examine these more closely.

In the macroscopic range, the width of transfer spans many words. The inner loop across a horizontal scan line gets executed many times, and the operations requested tend to be simple moves or constant stores. Examples of these are:

- Clearing a line of text to white
- Clearing an entire window to white
- Scrolling a block of text up or down

It is fortuitous that most processors provide a fast means for block moves and stores, and these can be made to serve the applications above. Suppose we structure the horizontal loop of BitBlt as the following sequence:

1. Move left partial word
2. Move many whole words (or none)
3. Move right partial word (or none)

Special cases can be provided for item 2 if the operation is a simple store or if it is a simple copy with no skew (horizontal bit offset) from source to destination. In this way, most macroscopic applications of BitBlt can be made fast, even on processors of modest power.

The microscopic range of BitBlt is characterized by a zero count for the inner loop in item 2, so that the work on each scanline involves, at most, two words. Both overall setup and vertical loop overhead can be considerably reduced for this case. Because characters tend to be less than a word wide and lines tend to be less than a word thick, nearly all text and line drawing fall into this category. A convenient way to provide such efficiency is to write a special case of BitBlt that assumes the microscopic parameters, but goes to the general BitBlt whenever these are not met. Because of the statistics (many small operations and a few very large ones), it does not hurt to pay the penalty of a false assumption on infrequent calls.■

Building Data Structures in the Smalltalk-80 System

James C Althoff Jr
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Most programmers are exposed to the concept of data structures very early in their programming experience. A course in data structures is an integral part of most computer science curricula, and there are many excellent and widely used texts on the subject (see references 1, 2, and 4). The data structures covered in these texts generally include the linear list, stack, queue, tree, and graph.

In this article, we will define and implement some of the simplest structures, including the linear list, stack, and queue. Our approach will be to describe each data structure informally, and then to show a Smalltalk-80 class definition that closely matches this informal description. We will see that it is possible, using the class construct, to create programming structures that clearly mirror the entities being implemented. However, in order to demonstrate how to build these data structures from scratch, we will not make use of any of the advanced data structure classes that already exist in the Smalltalk-80 system.

We will make extensive use of the Smalltalk-80 subclass mechanism in the class definitions we introduce. We will use subclassing to facilitate the construction of different implementations of the same entity. In addition, we will see how the subclass mechanism enables us to define two or more related classes in such a way that the common parts of their definition can be shared.

Notation

In order to understand the program examples presented in this article, the reader should have some familiarity with the Smalltalk-80 programming language. For an introduction to the language and a fuller discussion of subclasses, see "The Smalltalk-80 System," on page 36. In addition, a text box on page 240 of this article contains a brief description of some of the messages that we will use in the examples.

The advantage of the sequential list is that it is easy to access and replace an arbitrary item in the list.

For each data structure that we describe, we will give a corresponding Smalltalk-80 class definition. Each class definition will be presented in the form of a template that shows the instance variables, messages, and other information associated with the class. (A complete description of the template can be found in the article mentioned above.)

The messages defined in the template are partitioned into two main groups. The first comprises the class messages. These are messages that are sent to the class itself (which is, in actuality, an object). For our purposes, these will be a set of creation messages that can be sent to the class in order to create a new, initialized instance of the class. The second group consists of a set of messages that can be sent to instances of the class. These will be divided into two more groups. The first is a collection of external messages that represent the interface between an instance of the class and clients (ie: other objects in the system) of that instance. The second is a set of internal messages used in methods that are defined in the class or a subclass but are not intended for wider use. Note that the distinction between internal and external messages is made for conceptual clarity. The Smalltalk-80 programming language does not have a mechanism for controlling message usage.

An example of a class definition is given in table 1. This template defines a class whose name is Card. An instance of class Card can be used to represent a card in a game program. Class Card has instance variables named suit, rank, and faceUp. A new instance is created by sending class Card the creation message suit:rank:. For example:

```
|aCard|
aCard ← Card suit: 'heart' rank: 7.
```

creates a new instance of Card that represents the seven of hearts. In the method for suit:rank:, the message new creates an uninitialized instance. The internal message setSuit:setRank: sets the suit and rank fields and initializes the new instance to be "face down." Given an instance of class Card, we can determine its suit, rank, and orientation, and change the latter using the external messages specified in the class definition. Because we do not want to be able to change the suit and rank of an instance once it has been created, we do not include a message for doing this operation in the set of external messages.

The methods shown here demonstrate a convention we will use in subsequent examples. All of the names that we use for parameters and local variables will be formed by taking the name of a class and preceding it with an indefinite article. For example, the two parameters to the message setSuit:setRank: are named aString and anInteger. Such a name indicates what kind of object is expected as the value of the parameter or local variable to which the name refers. Smalltalk-80 has no type-checking; this is only a convention to help make the examples more understandable.

We will now examine a number of elementary data structures and their implementation with Smalltalk-80 classes. In the informal description of each data structure, we will include a list of operations that are meaningful for that structure. We will then show a corresponding class definition whose external messages match the listed operations. The details of the implementation of the data structure will be hidden in the class definition. We will see how this process enables us to define different classes that reflect different implementations of the same basic data structure.

The Linear List

The first data structure we will examine is the *linear list*. A linear list is a sequence of data items that have, essentially, a one-dimensional relationship to one another (see figure 1). That is to say, given an object in the sequence, we can find the object that precedes or follows it. For example, if we have a program that deals with a game of cards, we might represent

class name	Card
superclass	Object
instance variable names	suit rank faceUp
class messages and methods	

```
suit: aString rank: anInteger | aCard |
   aCard ← self new. aCard setSuit: aString setRank: anInteger.
   ↑ aCard.
```

instance messages and methods

```
external
suit | | ↑ suit.
rank | | ↑  rank.
turnFaceUp | | faceUp ← true.
turnFaceDown | | faceUp ← false.
turnOver | | faceUp ← faceUp not.
isFaceUp | | ↑ faceUp.
isFaceDown | | ↑ faceUp not.

internal
setSuit: aString setRank: anInteger | |
   suit ← aString. rank ← anInteger. self turnFaceDown.
```

Table 1: *Class template for class* Card.

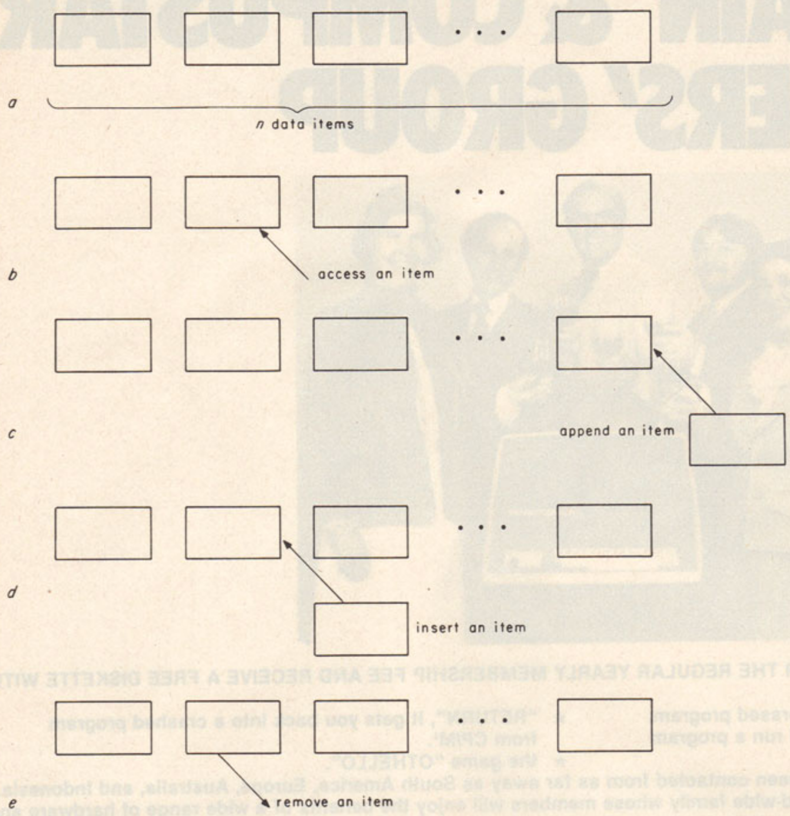


Figure 1: *A linear list (1a) is a collection of objects arranged in linear sequence. Permissible operations include accessing (1b), appending (1c), inserting (1d), and removing an item (1e).*

each card hand as a linear list of cards.

The operations that we might want to perform on a linear list include:

- determine how many items are in the list
- determine whether or not the list is empty
- access the *i*th item in the list
- append an item to the end of the list
- insert an item at some position in the list
- replace an item at some position in the list with some other item
- remove an item from the list

In order to implement a linear list of data items, we need to implement both the data items and the linear list. In all of our subsequent examples, we will assume that we have implemented the data items with one or more class definitions (eg: class Card). When we are describing things in general, we will use such terms as "data item" and "linear list." When we are describing a specific implementation, we will refer instead to the "object" (or "instance of a class") that represents the entity under consideration.

Linear lists are partly defined in the template given in table 2. Class LinearList, as defined in table 2, is incomplete since there is no mechanism for actually storing objects that represent data items. This is because there are several different strategies for storing these objects in a linear list. For each strategy, we will define a different class, each of which is a subclass of LinearList. All of these classes have some characteristics in common; these are captured in the superclass LinearList. For example, because all of the subclasses that we

class name	LinearList
superclass	Object
instance variable names	count
class messages and methods	

```
"none defined here"
```

instance messages and methods

```
external
count | | ↑ count.
empty | | ↑ count = 0.
"at: anInteger | | ... to be defined in subclasses"
"append: anObject | | ... to be defined in subclasses"
"insert: anObject at: anInteger | | ... to be defined in subclasses"
"replace: anObject at: anInteger | | ... to be defined in subclasses"
"removeAt: anInteger | | ... to be defined in subclasses"

internal
initialize | | count ← 0.
checkIndex: anInteger | |
   (anInteger < 1) | (anInteger > count)
   ifTrue: [ ↑ self error: 'index out of range' ].
```

Table 2: *Class template for class* LinearList.

will define keep count of the number of objects (representing data items) in the list, a corresponding instance variable, count, is defined in the superclass LinearList. Similarly, all subclasses can make use of an internal message, initialize, that initializes count to zero, and another message, checkIndex:, that insures that any index specified as a parameter to one of the access messages (at:, insert:at:, and so on) is within range. Since we start the numbering of items in a linear list at 1, the range will always be between 1 and count (the number of items in the list). Also, the message empty is implemented in the superclass since the answer to whether or not the list is empty can be determined from the value of count.

The Indexed Table

The first strategy we will explore for actually storing data items in a linear list involves the sequential allocation of storage. In order to see how this works, we will interrupt our discussion of the linear list and introduce a very basic data structure that we will call an *indexed table* (see figure 2). An indexed table, which corresponds to what is called a one-dimensional array in many programming languages, is a relatively simple structure that closely matches the physical memory of most computers. As we will see, many useful data structures, including the linear list, can be implemented with an indexed table.

An indexed table comprises a sequence of variables into which we can store and from which we can retrieve data items. Each variable is designated by an integer. The smallest integer used to designate a variable in an indexed table is called the *lower bound* of the table. The largest integer is called the *upper bound*. The operations that we wish to perform on an indexed table are:

- specify the lower and upper bounds of the table (when it is created)
- determine the lower and upper bounds of the table
- determine the number of variables allocated to the table

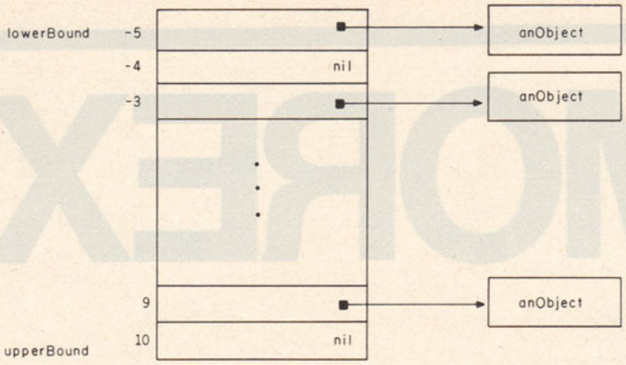


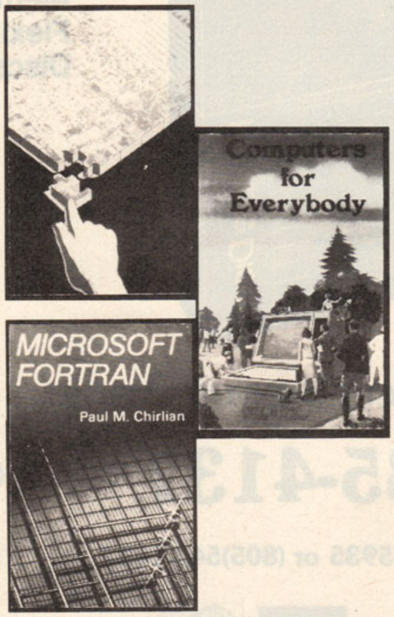
Figure 2: *An indexed table.*

- access an item at a particular position in the table
- put an item at a particular position in the table

In order to store into an indexed table, we specify an item to be stored and an integer that indicates the particular variable that will contain the item. In order to retrieve from an indexed table, we need only specify which variable of the table has the data item of interest.

The class definition in table 3 shows how to implement an indexed table. We have seen in the definition of classes Card and LinearList how to specify a fixed number of named instance variables in a class definition. What we need in order to implement an indexed table, however, is a sequence of unnamed variables that are designated by an integer index. How can we define such a sequence of variables? We do this by specifying the number of indexed variables needed for an instance at the time that an instance of a class is created. The creation message from:to: in class IndexedTable sends the message new:, whose parameter is the number of indexed variables required for the instance being created. In order to access these variables, we send low-level (ie: primitive) access messages with a parameter that specifies an appropriate index (starting at 1). The access message basicAt: *i* retrieves the object stored in the *i*th indexed instance variable. The access message basicAt: *i* put: anObject stores a pointer to anObject in the *i*th indexed instance variable.

In addition to indexed instance variables, class IndexedTable has two named instance variables: lowerBound and upperBound. lowerBound is an integer that indicates the smallest allowable index for a particular instance of IndexedTable; upperBound indicates the largest allowable index. In order to create a new indexed table, we send the message



class name	IndexedTable
superclass	Object
instance variable names	lowerBound upperBound
class messages and methods	

```
from: anInteger1 to: anInteger2 | |
    (anInteger1 > anInteger2) ifTrue: [ ↑ self error: 'invalid bounds' ].
    ↑ (self new: anInteger2 − anInteger1 + 1)
    lowerBound: anInteger1 upperBound: anInteger 2.
```

instance messages and methods

```
external
lowerBound | | ↑ lowerBound.
upperBound | | ↑ upperBound.
size | | ↑ upperBound − lowerBound + 1.
at: anInteger | |
    self checkIndex: anInteger.
    ↑ self basicAt: (anInteger − lowerBound + 1).
put: anObject at: anInteger | |
    self checkIndex: anInteger.
    self basicAt: (anInteger − lowerBound + 1) put: anObject.

internal
lowerBound: anInteger1 upperBound: anInteger2 | |
    lowerBound ← anInteger1. upperBound ← anInteger2.
checkIndex: anInteger | |
    (anInteger < lowerBound) | (anInteger > upperBound)
    ifTrue: [ ↑ self error: 'index out of range' ].
"basicAt: i | | ... this is a primitive Smalltalk-80 message that accesses the ith indexed
  instance variable."
"basicAt: i put: anObject | | ... this is a primitive Smalltalk-80 message that stores a pointer
  to anObject in the ith indexed instance variable."
```

Table 3: *Class template for class* IndexedTable.

from:to: to class IndexedTable. For example:

```
| table |
table ← IndexedTable from: −5 to:
  10.
```

creates a new instance of IndexedTable whose indices range from −5 to 10. This message is implemented using an internal message, lowerBound:upperBound:, that sets the instance variables of a newly created instance to their appropriate values.

Once we have created an instance of IndexedTable, we can perform the operations specified in the above description of indexed tables by sending the messages lowerBound, upperBound, size, at:, and put:at:. lowerBound and upperBound return the corresponding values of the instance variables; size computes and returns the number of indexed instance variables in the table. at: and put:at: both use an internal message, checkIndex:, in order to make sure that their index parameter is within range of the lower and upper bounds of the table. at: returns the object stored in the indexed instance variable indicated by the integer parameter; put:at: stores the object, specified as the first parameter, in the indexed instance variable indicated by the second parameter. Notice that both at: and put:at: use the value of lowerBound to map indices from the range of the table to the range of the indexed instance variables that are used to implement the table.

As an example of how we might use class IndexedTable, consider the following sequence of messages:

```
| table |
table ← IndexedTable from: 1 to:
  13.
1 to: 13 do: [: i | table put: (Card
  suit: 'heart' rank: i) at: i. ].
```

This creates a new instance of class IndexedTable with indices that range from 1 to 13 and fills it with instances of class Card whose ranks match the indices.

The Sequential List

Now that we have an implementa-

Some Smalltalk Messages

The following messages are used in this article without having been previously defined. Each is either provided by the Smalltalk-80 system, or easily implemented using other messages provided by the system. For each message, we provide a brief, informal description of its intended effect.

new—*Creates a new instance.*
new: i—*Creates a new instance with* i *indexed instance variables.*
error: aString—*Causes some appropriate action to occur, such as interrupting program execution and displaying* aString.
not—*The "logical not" operation.*
| —*The "logical or" operation.*
+, −, *, /, =, <, >, <=—*Arithmetic operations and relations.*

Flow of control is affected by sending messages that correspond to basic control structures. For example:

a ifTrue: [b] *corresponds to:* if a then b end
a ifTrue: [b] ifFalse: [c] *corresponds to:* if a then b else c end
x to: y do: [: i | a] *corresponds to:* for i from x to y by 1 repeat a end
x downTo: y do: [: i | a] *corresponds to:* for i from x to y by −1 repeat a end
x timesRepeat: [a] *corresponds to:* repeat x times a end

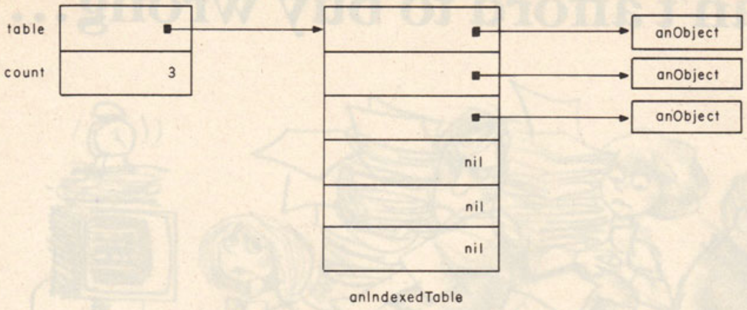


Figure 3: *A sequential list.*

tion for indexed tables, we can use them to demonstrate our first implementation of linear lists. We will call a linear list that uses an indexed table to store its data items a *sequential list* (see figure 3). The basic idea is to manage an indexed table so that the first *i* consecutive entries in the table are the *i* data items in our linear list. The most difficult operations using this strategy are the insertion and removal of items, since these cause parts of the indexed table to be copied from one area to another. An additional difficulty is that we must specify, at the time we create the linear list, the expected maximum number of items in the list. This is necessary because indexed tables come in fixed sizes, which means that if the list grows larger than this initial number, we must do something to accommodate the extra items. (Details are shown in the class definition given in table 4.)

The creation message size: is used to create an instance of SequentialList of some estimated maximum size. For example:

```
| list |
list ← SequentialList size: 5.
```

creates a list with enough space, initially, for five objects. The internal message initialize:, which is sent from size:, creates an instance of IndexedTable of the appropriate size and assigns it as the value of the instance variable named table. The other internal message, expand, is used to enlarge table when it becomes full. This is done by creating a new instance of IndexedTable that is twice as large as the original and by copying the objects from the original table into the first half of the new table. The new table is then assigned as the value of table for subsequent use.

Retrieval from SequentialList is done by retrieving from its associated indexed table. A new object can be appended to the list by storing it in the next available location of the indexed table of that list. This location is determined by the value of count. If there is still room in the table, count is incremented and the object is stored. Otherwise, the sequential list has to be expanded, which is accomplished using the message expand, described previously. Insertion into the list is done by copying from their current location to the next, all objects after, and including, the one at the desired location. A new object can then be stored at that location. Removing an object from a list is done analogously. An object is replaced by storing another object in the corresponding position in the indexed table.

The advantage of the sequential list is that it is easy to access and replace (not remove) an arbitrary item. The disadvantages are that it is necessary both to estimate the maximum size of the list when it is created (although, as we have seen, the list can expand when necessary) and to move items around when inserting or removing them from the list.

Let us now consider an example that shows how to create and send messages to an instance of class SequentialList. Suppose we want to represent a deck of cards and two hands, dealt from the deck. To create two initially empty hands we write:

```
| hand1 hand2 deck |
hand1 ← SequentialList size: 5.
hand2 ← SequentialList size: 5.
```

To create an unshuffled deck of fifty-two cards we write:

```
deck ← SequentialList size: 52.
1 to: 13 do: [: i | deck append:
  (Card suit: 'heart' rank: i). ].
1 to: 13 do: [: i | deck append:
  (Card suit: 'diamond' rank: i). ].
1 to: 13 do: [: i | deck append:
  (Card suit: 'club' rank: i). ].
1 to: 13 do: [: i | deck append:
  (Card suit: 'spade' rank: i). ].
```

Then, to deal five cards from the deck to the first hand we write:

```
5 timesRepeat: [ hand1 append:
   (deck removeAt: 1). ].
```

To deal from the bottom of the deck to the second hand we write:

```
5 timesRepeat: [ hand2 append:
   (deck removeAt: deck
   count). ].
```

The Linked List

A second approach for managing the storage of items in a linear list is to use a *linked list*. The strategy for

class name	SequentialList
superclass	LinearList
instance variable names	table
class messages and methods	

```
size: anInteger | |
    (anInteger > 0) ifTrue:   [ ↑ (self new) initialize: anInteger ]
                    ifFalse:  [ ↑ self error: 'invalid size' ].
```

instance messages and methods

```
external
at: anInteger | | self checkIndex: anInteger. ↑ table at: anInteger.
append: anObject | |
    (count = table size) ifTrue: [ self expand. ].
    table put: anObject at: count + 1.
    count ← count + 1.
insert: anObject at: anInteger | |
    self checkIndex: anInteger.
    (count = table size) ifTrue: [ self expand. ].
    count downTo: anInteger do: [: i | table put: (table at: i) at: i + 1. ].
    table put: anObject at: anInteger.
    count ← count + 1.
replace: anObject at: anInteger | |
    self checkIndex: anInteger.
    table put: anObject at: anInteger.
removeAt: anInteger | anObject |
    anObject ← self at: anInteger.
    anInteger + 1 to: count do: [: i | table put: (table at: i) at: i − 1. ].
    count ← count − 1.
    ↑ anObject.

internal
initialize: anInteger | |
    super initialize.
    table ← IndexedTable from: 1 to: anInteger.
expand | anIndexedTable |
    anIndexedTable ← IndexedTable from: 1 to: 2 * count.
    1 to: count do: [: i | anIndexedTable put: (table at: i) at: i. ].
    table ← anIndexedTable.
```

Table 4: *Class template for class* SequentialList.

using a linked list is the following: instead of allocating sequentially the storage needed to hold data items, we allocate separate storage objects, called *links*, each of which keeps track of a data item and either one or two other links. A set of links are connected together to form a linked list. A *single link* (see figure 4a) is one that keeps track of a data item and one other link, which is its successor. A *double link* (see figure 4b) has a data item and two other links: its successor and its predecessor.

Class SingleLink is defined as shown in table 5. It has instance variables named entry and successor; entry points to an object that represents a data item, successor points either to another instance of class SingleLink, or, if there is no successor, to the object nil. We specify the entry and successor of a single link at the time we create it. For example:

```
| link1 link2 |
link1 ← SingleLink entry: (Card
    suit: 'club' rank: 4) successor:
    nil.
link2 ← SingleLink entry: (Card
    suit: 'diamond' rank: 6)
    successor: link1.
```

creates two links. The entry of the first link is an instance of class Card that represents the four of clubs. Its successor is nil. The entry of the second link is an instance of class Card that represents the six of diamonds; its successor is the first link.

The class definition for double links is given in table 6. Class DoubleLink inherits from class SingleLink the instance variables and messages that are used to implement the entry and successor of a double link. In addition, there is an instance variable named predecessor that points either to an instance of class DoubleLink or to nil. The messages predecessor: and predecessor set and return, respectively, the value of predecessor.

As stated previously, a linked list is a sequence of links connected in a linear arrangement. We can make different kinds of linked lists depending on the links we use and the precise

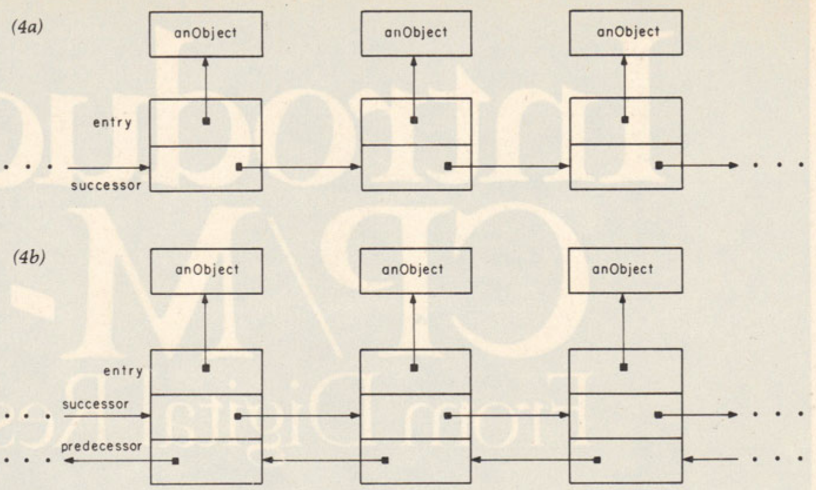


Figure 4: *Two kinds of links. Figure 4a shows a list of storage objects joined by single links, while figure 4b shows a similar list joined by double links.*

class name	SingleLink
superclass	Object
instance variable names	entry successor
class messages and methods	
	entry: anObject successor: aSingleLink | | ↑ ((self new) entry: anObject) successor: aSingleLink.
instance messages and methods	
	external **entry** | | ↑ entry. **entry: anObject** | | entry ← anObject. **successor** | | ↑ successor. **successor: aSingleLink** | | successor ← aSingleLink. *internal* "none defined here"

Table 5: *Class template for class* SingleLink.

way that they are connected together. Table 7 defines class LinkedList, which acts as the superclass of the various linked lists we will consider. It collects several messages that are suitable for all of the LinkedList subclasses.

The message linkAt: is used internally by LinkedList methods. Taking an integer as a parameter, it traverses a sequence of connected links looking for the link in the position indicated by that integer. It returns this link as a result. This message is not intended for use outside of the class since we don't want the rest of the system to have access to the internal structure of the list. The messages at: and replace:at: can be defined using linkAt: since, once the appropriate link is found, it is easy to return or replace its corresponding entry.

The Singly Linked List

The first type of linked list we will examine is the *singly linked list*. A singly linked list is a sequence of single links connected together so that the successor of the first link is the second link, the successor of the second link is the third link, and so on (see figure 5). The successor of the last link is nil.

The definition for class SinglyLinkedList is given in table 8. Class SinglyLinkedList has an instance variable named firstLink that points either to the first object in the list or to nil if the list is empty. The internal message initialize, which is sent when an instance of SinglyLinkedList is created, sets firstLink to nil. The internal message firstLink, which is never sent to an empty list, returns the SingleLink instance pointed to by firstLink.

In the implementation of the message append:, we first create a new link whose entry is the object passed as a parameter. We then check

class name	DoubleLink
superclass	SingleLink
instance variable names	predecessor
class messages and methods	

```
entry: anObject successor: aDoubleLink1 predecessor:
   aDoubleLink2 | |
    ↑ (self entry: anObject successor: aDoubleLink1) predecessor:
      aDoubleLink2.
```

instance messages and methods	

```
external
predecessor | | ↑ predecessor.
predecessor: aDoubleLink | | predecessor ← aDoubleLink.

internal
"none defined here"
```

Table 6: *Class template for class* DoubleLink.

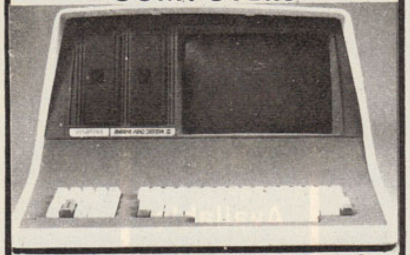
class name	LinkedList
superclass	LinearList
instance variable names	"none defined here"
class messages and methods	

```
new | | ↑ (super new) initialize.
```

instance messages and methods	

```
external
at: anInteger | |
    self checkIndex: anInteger.
    ↑ (self linkAt: anInteger) entry.
replace: anObject at: anInteger | |
    self checkIndex: anInteger.
    (self linkAt: anInteger) entry: anObject.

internal
"firstLink | | ... to be defined in subclasses"
linkAt: anInteger | aLink |
    aLink ← self firstLink.
    (anInteger − 1) timesRepeat: [ aLink ← aLink successor. ].
    ↑ aLink.
```

Table 7: *Class template for class* LinkedList.



to see if the list is empty, and if so we set firstLink to point to the new link. If the list is not empty, we get the last link in the list and make its successor the new link. In either case, we increment the value of count to keep track of the new number of objects in the list.

In general, to insert a data item into a singly linked list, we create a new link for the item and then find the link that the new link is to follow. We then make this link point to the new link, and have the new link point to the former successor of this link (see figure 6a). The only exception to this process occurs if we are adding the data item to the beginning of the list. In this case there is no predecessor, so we simply point the new link to the

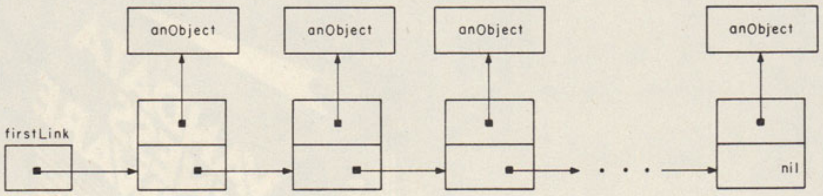
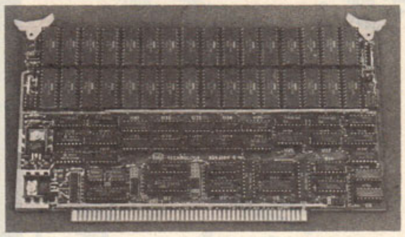


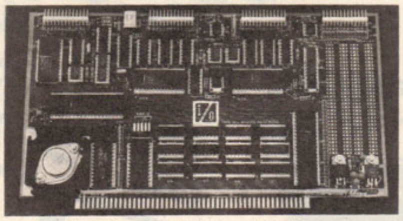
Figure 5: *A singly linked list.*

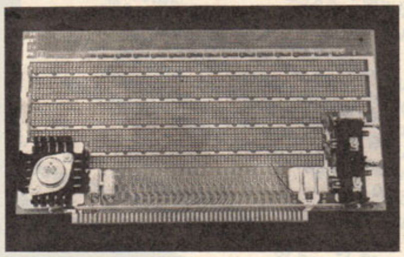
class name	SinglyLinkedList
superclass	LinkedList
instance variable names	firstLink
class messages and methods	
"none defined here"	
instance messages and methods	

```
external
append: anObject | aSingleLink |
    aSingleLink ← SingleLink entry: anObject successor: nil.
    self empty
        ifTrue: [ firstLink ← aSingleLink. ]
        ifFalse: [ (self linkAt: count) successor: aSingleLink. ].
    count ← count + 1.
insert: anObject at: anInteger | aSingleLink |
    self checkIndex: anInteger.
    (anInteger = 1)
        ifTrue: [ firstLink ← SingleLink entry: anObject successor: firstLink. ]
        ifFalse: [ aSingleLink ← self linkAt: anInteger − 1.
            aSingleLink successor: (SingleLink entry: anObject successor:
            aSingleLink successor). ].
    count ← count + 1.
removeAt: anInteger | aSingleLink anObject |
    self checkIndex: anInteger.
    (anInteger = 1)
        ifTrue: [ anObject ← firstLink entry. firstLink ← firstLink successor. ]
        ifFalse: [ aSingleLink ← self linkAt: anInteger − 1.
            anObject ← aSingleLink successor entry.
            aSingleLink successor: aSingleLink successor successor. ].
    count ← count − 1.
    ↑ anObject.

internal
initialize  |  | super initialize. firstLink ← nil.
firstLink   |  | ↑ firstLink.
```

Table 8: *Class template for class* SinglyLinkedList.







former first link in the list. This procedure is used to implement the message insert:At: in class SinglyLinkedList. We also increment the value of count, just as we do in the message append.

The general procedure for removing a data item from a singly linked list is analogous to that for inserting an item. First we find the link that precedes the one at the position of interest. We then point this link to the link that follows the one of interest. This deletes the link of interest from the list (see figure 6b). We then return the entry of the deleted link. Again, the exceptional case is removing the first item since there is no preceding link. The message removeAt: in class SinglyLinkedList is implemeneted using this procedure. Since an object has been deleted from the list, we decrement the value of count.

Suppose we wish to use a singly linked list instead of a sequential list in our previous example of a deck of cards. We simply create instances of class SinglyLinkedList for the deck and hands, as shown in the following:

```
| hand1 hand2 deck |
deck ← SinglyLinkedList new.
hand1 ← SinglyLinkedList new.
hand2 ← SinglyLinkedList new.
```

We can then use the rest of the code, unchanged, from that example. We are able to do this because we have hidden the details of each implementation inside the corresponding class definition and, in using the classes, have limited ourselves to a clearly defined set of external messages.

The Circular List

Another type of linked list is the *circular list* (see figure 7). A circular list is a singly linked list in which the successor of the last link in the list is the first link in the list. This makes a circular chain of links. If we have access to the last link in a circular list, then we also have easy access to the first link of that list since it is the immediate successor of the last link. By keeping track of the last link of a circular list, we can easily insert and remove items from both the begin-

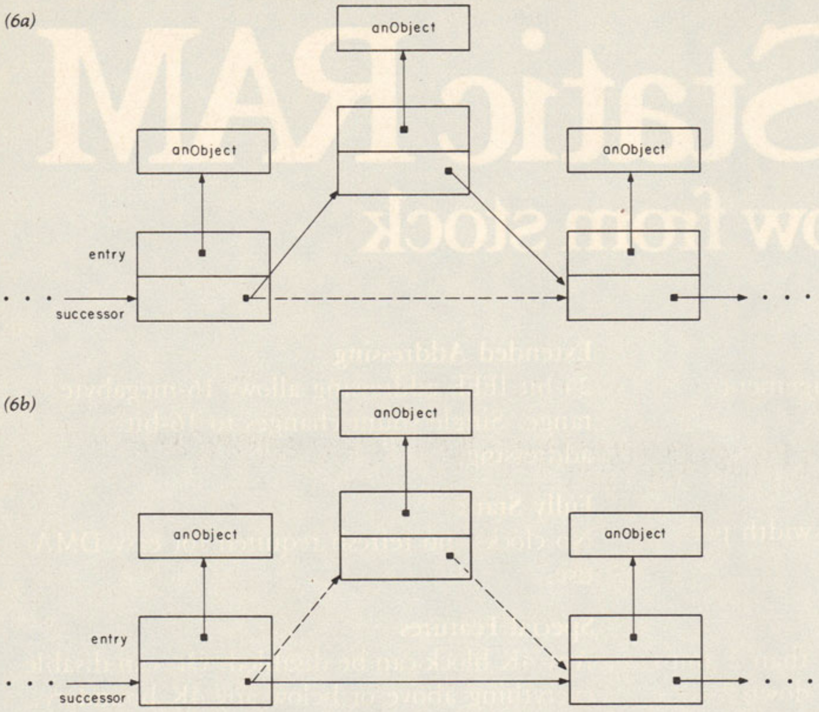


Figure 6: *The insertion and deletion of data items from a singly linked list. In both insertion (6a) and deletion (6b) processes, the dotted lines represent the links existing before the process, while the solid lines represent the links existing after the process.*

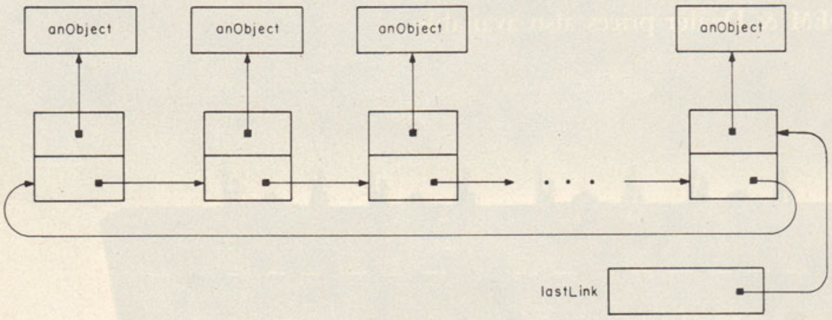


Figure 7: *A circular list.*

ning and the end of the list. This will be a useful property in some of the data structures that we will build out of circular lists.

The definition of class CircularList is given in table 9. Class CircularList has an instance variable named lastLink that points either to the last link in the list or to nil if the list is empty. The internal message initialize sets lastLink to nil. The internal message firstLink (again, as in the case of class SinglyLinkedList, sent to nonempty lists only) returns the successor of lastLink. Since the list is circular, this is the first link in the list.

The append, insert, and remove operations on a circular list are similar to those on a noncircular list. Since we always have a link that precedes any given link in a circular list, we don't have to make exceptions for operations on the beginning of the list. The implementation of these operations is demonstrated in the methods for the messages append:, insertAt:, and removeAt:.

The Doubly Linked List

Now we will consider the *doubly linked list* (see figure 8). A doubly linked list is a sequence of double links connected together. The successor of a given link is the link that follows it in the sequence, just as in the case of the singly linked list. The predecessor of a given link is the link that precedes it in the sequence. A doubly linked list can be made into a circular list, just as a singly linked list can, by connecting the first and last links. In this case, the successor of the

Text continued on page 260

class name	CircularList
superclass	LinkedList
instance variable names	lastLink
class messages and methods	

"none defined here"

instance messages and methods

```
external
append: anObject | aSingleLink |
   self empty
      ifTrue: [ lastLink ← SingleLink entry: anObject successor: nil.
         lastLink successor: lastLink. ]
      ifFalse: [ aSingleLink ← SingleLink
         entry: anObject successor: lastLink successor.
         lastLink successor: aSingleLink. lastLink ← aSingleLink. ].
   count ← count + 1.
insert: anObject at: anInteger | aSingleLink |
   self checkIndex: anInteger.
   aSingleLink ← self linkAt: anInteger − 1.
   aSingleLink successor:
   (SingleLink entry: anObject successor: aSingleLink successor).
   count ← count + 1.
removeAt: anInteger | aSingleLink anObject |
   self checkIndex: anInteger.
   aSingleLink ← self linkAt: anInteger − 1.
   anObject ← aSingleLink successor entry.
   (count = 1)
      ifTrue: [ lastLink ← nil. ]
      ifFalse: [ aSingleLink successor: aSingleLink successor successor.
         (anInteger = count) ifTrue: [ lastLink ← aSingleLink. ]. ].
   count ← count − 1.
   ↑ anObject.

internal
initialize | | super initialize. lastLink ← nil.
firstLink | | ↑ lastLink successor.
linkAt: anInteger | |
   (anInteger = count) | (anInteger = 0) ifTrue: [ ↑ lastLink ].
   ↑ super linkAt: anInteger.
```

Table 9: *Class template for class* CircularList.

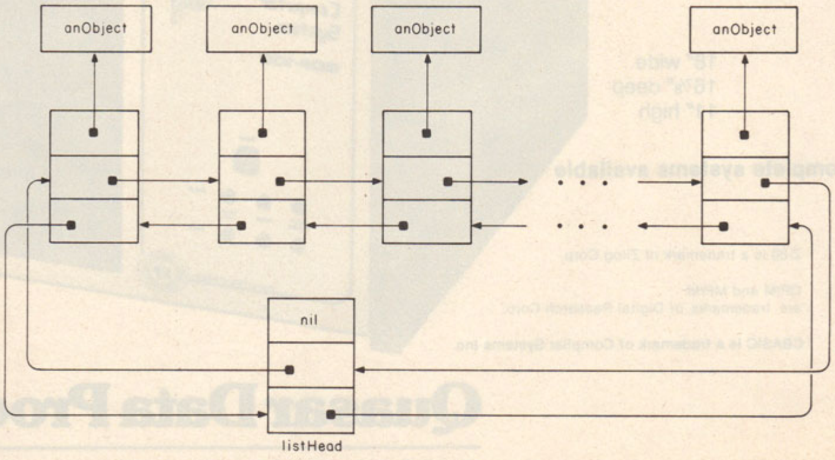


Figure 8: *A doubly linked list.*

class name	DoublyLinkedList
superclass	LinkedList
instance variable names	listHead
class messages and methods	
"none defined here"	
instance messages and methods	

```
external
append: anObject | | self insert: anObject after: listHead predecessor.
insert: anObject at: anInteger | |
   self checkIndex: anInteger.
   self insert: anObject after: (self linkAt: anInteger − 1).
removeAt: anInteger | aDoubleLink |
   self checkIndex: anInteger.
   aDoubleLink ← self linkAt: anInteger.
   aDoubleLink successor predecessor: aDoubleLink predecessor.
   aDoubleLink predecessor successor: aDoubleLink successor.
   ↑ aDoubleLink entry.

internal
initialize | |
   super initialize.
   listHead ← DoubleLink entry: nil successor: nil predecessor: nil.
   listHead successor: listHead. listHead predecessor: listHead.
firstLink | | ↑ listHead successor.
linkAt: anInteger | aDoubleLink |
   (anInteger = 0) ifTrue: [ ↑ listHead ].
   (anInteger < = (count / 2)) ifTrue: [ ↑ super linkAt: anInteger ].
   aDoubleLink ← listhead.
   (count − anInteger + 1) timesRepeat:
   [ aDoubleLink ← aDoubleLink predecessor. ].
   ↑ aDoubleLink.
insert: anObject after: aDoubleLink1 | aDoubleLink2 |
   aDoubleLink2 ← DoubleLink entry: anObject
      successor: aDoubleLink1 successor predecessor: aDoubleLink1.
   aDoubleLink1 successor: aDoubleLink2.
   aDoubleLink2 successor predecessor: aDoubleLink2.
```

Table 10: *Class template for class* DoublyLinkedList.

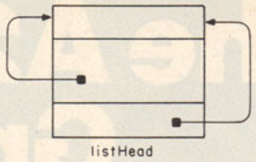


Figure 9: *An empty doubly linked list.*

Text continued from page 256:

last link is the first link, and the predecessor of the first link is the last link. The class definition for a circular, doubly linked list is given in table 10.

Class DoublyLinkedList has an instance variable named listHead, which points to a special kind of link known as a *list head*. A list head is a link whose entry is unused. The idea is to keep the list head in the list so that even when there are no data items in the list, at least one link is present. Having a link present at all times simplifies the implementation of some linked list operations. A list head can be used in the implementation of a singly linked list, but it is especially convenient in the implementation of a circular doubly linked list. In a circular doubly linked list, the list head successor points to the first link in the list (excluding the list head itself), or to itself if the list is empty (see figure 9). The list head predecessor points to the last link in the list (excluding the list head itself), or to itself if the list is empty. In class DoublyLinkedList, the internal message initialize sets listHead to an instance of DoubleLink whose entry is nil and whose successor and predecessor are both that same instance (ie: the double link is made circular). The message firstLink has been modified to return the list head successor.

Because a doubly linked list is composed of a sequence of double links, it is possible to traverse the list in both directions, forward and backward, with equal facility. The internal message linkAt: in class DoublyLinkedList has been modified to access objects past the middle of the list by starting from the rear and traversing toward the front. This improves the performance of the access message at:.

Inserting an item in a doubly linked list is similar to inserting an item in a singly linked list. We first create a new link for the item. Then we find the link that this new link is to follow, set the successor and predecessor pointers of the new link, and adjust both the successor pointer of the link that precedes the new link and the predecessor pointer of the link that follows the new link (see figure 10a). If we are using a circular list with a list head, we don't have to consider any exceptional cases. Removing an item from a list is an analogous process (see figure 10b). The details of these procedures are demonstrated in the methods for the messages append:, insert:, and removeAt:.

The Stack

The next data structure that we will look at is the *stack* (see figure 11). A stack is a linear list of items that is accessed in a very restricted way. In fact, only one side of a stack, the *top*, can be accessed. The *bottom* of the stack cannot be accessed. These names are useful because most stack diagrams list their items vertically, with the accessible end higher. The item at the accessible end of the stack

Text continued on page 264

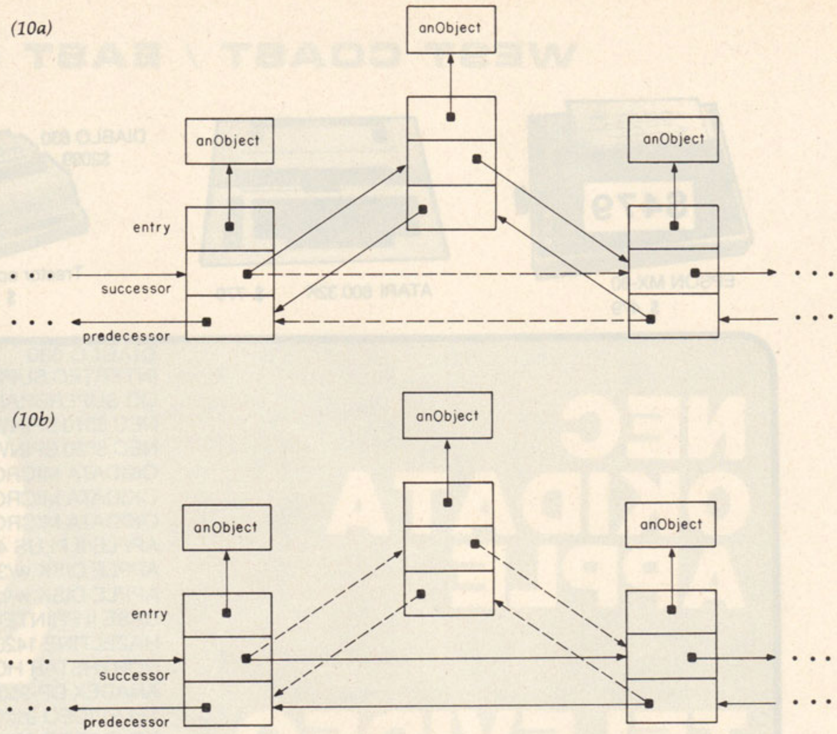


Figure 10: *The insertion and deletion of data items from a doubly linked list. In both insertion (10a) and deletion (10b) processes, the dotted lines represent the links existing before the process, while the solid lines represent the links existing after the process.*

Text continued from page 260:

is called the *top item*. A new item is added to the accessible end, thereby making it the new top item. This is called *pushing* an item onto a stack. Only the top item can be removed, or *popped*, from a stack. By adding and removing in this fashion (pushing and popping), we are able to access items in a last-in-first-out manner—that is, the last item pushed on a stack is the first item to be popped off the stack. Because of this, a stack is often called a *LIFO* (last-in-first-out).

Many examples of collecting and accessing in stack fashion exist outside the realm of programming. A pile of trays in a cafeteria rack is often used in this way. The same can be true of papers piled on a desk. In programming systems, a number of algorithms call for the use of a stack. For example, arithmetic expressions expressed in prefix or postfix notation can be evaluated using a stack to keep track of partial results.

The operations that we want to perform on a stack include:

- determine the number of items on the stack
- determine whether or not the stack is empty
- push an item onto the stack
- pop an item off the stack
- access the top item on the stack

It is easy to implement a stack using a linear list as the basic storage mechanism. Since we have several kinds of linear lists, it is possible to have several different stack implementations. Table 11 gives the definition of class Stack, which serves as a superclass for subsequent stack classes. Each kind of stack has a buffer which is a linear list, either sequential or linked. The messages count and empty are implemented using the corresponding messages of the linear list. The message emptyCheck, which sends an error message if the stack is empty, will be used in the implementations of the messages pop and top.

The Sequential Stack

The first stack implementation we

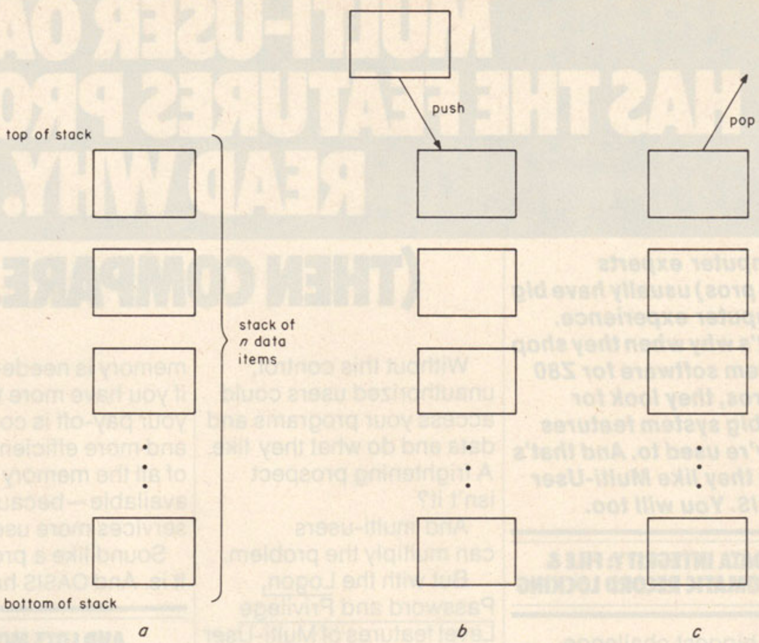


Figure 11: *A stack, shown in figure 11a, is a linear collection of objects arranged so that items can be added to or removed from the stack only at the top end of the stack. Figure 11b shows an item being added, or pushed, to the stack. Figure 11c shows an item being taken, or popped, from the stack.*

will consider is the *sequential stack*. A sequential stack is simply a stack implemented using a sequential list. The definition of class SequentialStack is given in table 12. An instance of SequentialStack is created by sending class SequentialStack the message size:, whose parameter is an integer indicating the estimated maximum size of the stack. For example:

```
| stack |
stack ← SequentialStack size: 20.
```

creates an instance of SequentialStack with space initially for twenty items. This size is expanded when necessary since the sequential list used as a buffer is expanded when required.

The message push: is implemented by appending to the buffer the object passed as a parameter. As we have noted, this is an easy operation for sequential lists to perform, except when the list overflows and requires expansion. The message pop is implemented by first checking to see if the buffer is empty, in which case an error is reported. Otherwise, the last item from the buffer is removed. Again, we should recall that this is an easy operation for a sequential list to perform. Similarly, in order to access the top of the stack (using the message top), we check to see that the stack is not empty, in which case we return (without removing) the last item in the buffer.

The following is an example using class SequentialStack:

```
| stack a b c |
stack ← SequentialStack size: 10.
a ← (Card suit: 'heart' rank: 5).
b ← (Card suit: 'heart' rank: 6).
c ← (Card suit: 'heart' rank: 7).
stack push: a.
stack push: b.
stack push: c.
a ← stack pop.
b ← stack pop.
c ← stack pop.
```

This example creates an instance of class SequentialStack that initially has space for ten objects. The variables a, b, and c are assigned to instances of class Card with ranks 5, 6, and 7, respectively. These instances are pushed on the stack and then popped

class name	Stack
superclass	Object
instance variable names	buffer
class messages and methods	"none defined here"
instance messages and methods	*external* **count** | | ↑ buffer count **empty** | | ↑ buffer empty "push: anObject | | ... to be defined in subclasses" "pop | | ... to be defined in subclasses" "top | | ... to be defined in subclasses" *internal* **emptyCheck** | | self empty ifTrue: [↑ self error: 'stack empty'].

Table 11: *Class template for class* Stack.

class name	SequentialStack
superclass	Stack
instance variable names	"none defined here"
class messages and methods	**size: anInteger** | | (anInteger > 0) ifTrue: [↑ (self new) initialize: anInteger] ifFalse: [↑ self error: 'invalid size'].
instance messages and methods	*external* **push: anObject** | | buffer append: anObject. **pop** | | self emptyCheck. ↑ buffer removeAt: buffer count. **top** | | self emptyCheck. ↑ buffer at: buffer count. *internal* **initialize: anInteger** | | buffer ← SequentialList size: anInteger.

Table 12: *Class template for class* SequentialStack.

off. The effect is to reverse the assignments to a, b, and c, such that the ranks are 7, 6, and 5, respectively.

The Linked Stack

Alternatively, we can define a *linked stack*, which is a stack whose buffer is a linked list. The definition of class LinkedStack is given in table 13. A linked stack is created by sending the message new to class LinkedStack. Since we are using a linked list for the buffer, there is no need to specify a maximum size estimate. For example:

```
| stack |
stack ← LinkedStack new.
```

creates a new instance of class

class name	LinkedStack
superclass	Stack
instance variable names	"none defined here"

class messages and methods

new | | ↑ (super new) initialize

instance messages and methods

external
push: anObject | | buffer insert: anObject at: 1.
pop | | self emptyCheck. ↑ buffer removeAt: 1.
top | | self emptyCheck. ↑ buffer at: 1.

internal
initialize | | buffer ← LinkedList new.

Table 13: *Class template for class* LinkedStack.

LinkedStack. The message push: is implemented by inserting the object passed as a parameter at the beginning of the buffer (ie: at position number 1). This is an easy operation for a singly linked list. The message pop is done by removing the first object from the buffer—another easy operation. The message top is implemented by accessing the object that is the entry of the first link of the buffer. We can use an instance of class LinkedStack in the example given for class SequentialStack by doing the following:

```
| stack |
stack ← LinkedStack new.
...
```

The rest of the example is unchanged.

The Queue

The *queue* is an important data structure that, like the stack, occurs often both in programming systems and outside the realm of programming (see figure 12). A queue is a linear list of items whose access is restricted to the two ends. An item can be appended to only one end of a queue, called the *rear*. An item can be removed only from the other end of the queue, called the *front* of the queue. This causes a sequence of items that are added to a queue and subsequently removed, to be accessed in a strict first-in-first-out fashion (ie: the first item that we put in a queue is the first item that we get out). Because of this, a queue is sometimes called a *FIFO* (first-in-first-out).

Lines of customers at a bank win-

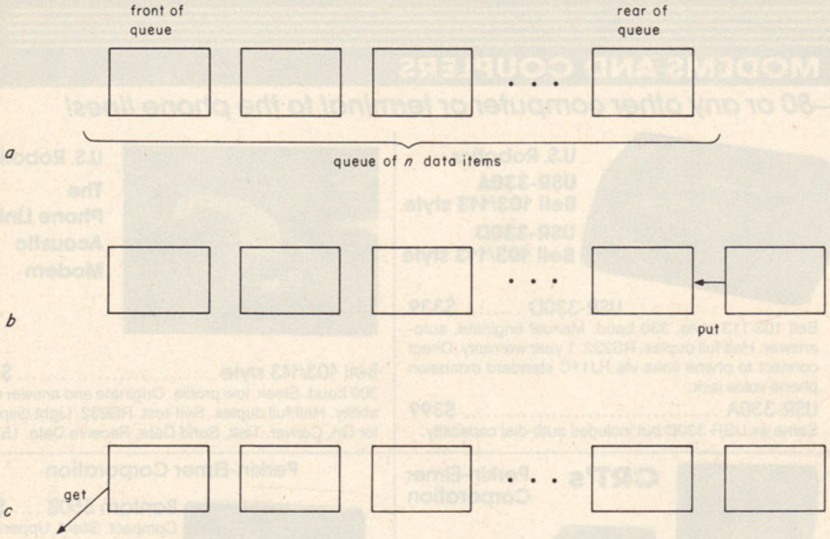


Figure 12: *A queue (12a) is a linear collection of objects arranged so that items can be added (or put) only at the rear of the queue (12b) and taken away (or gotten) only at the front of the queue (12c).*

dow or checkout counter are everyday examples of this kind of discipline. In programming systems, queues are used for many purposes, for example, to represent a line of customers in a simulation program or to handle ordered lists of events and processes in operating systems.

The operations we want to perform on a queue include:

- determine the number of items in the queue
- determine whether or not the queue is empty
- put an item in the queue
- get an item from the queue

Just as in the case of the stack, we can have different implementations of the queue depending on the type of buffer we use for storing the data items. Class Queue, which serves as a superclass for the queue classes, is given in table 14. Class Queue contains an instance variable named buffer that points to the object that provides the storage for items in the queue. The external messages have been listed to show what must be defined in all subclasses. The message emptyCheck, which sends an error message if the queue is empty, will be used in the implementation of get.

The Linked Queue

The first queue we will consider is the *linked queue*. A linked queue is one whose buffer is a linked list. Because we want to remove and add items easily, we will use a circular, singly linked list in our implementation. Class LinkedQueue is shown in table 15. An instance of class LinkedQueue is created by sending the creation message new to the class. For example:

```
  queue |
queue ← LinkedQueue new.
```

creates and initializes a new linked queue. The internal message initialize creates a new instance of CircularList and stores a pointer to it in buffer.

To put an object in a linked queue we use the message put:, passing the object as the parameter. This object is then added to the circular list by sending buffer the message append:. Similarly, the next object can be removed by using the message get. The method for get first checks to see if buffer is empty. If it is, an error message is sent. If not, the first object is removed from buffer and returned.

The messages count and empty are implemented by sending the respective messages to buffer and returning

<table>
<tr><td>class name</td><td>Queue</td></tr>
<tr><td>superclass</td><td>Object</td></tr>
<tr><td>instance variable names</td><td>buffer</td></tr>
<tr><td>class messages and methods</td><td></td></tr>
<tr><td>"none defined here"</td><td></td></tr>
<tr><td>instance messages and methods</td><td></td></tr>
<tr><td colspan="2"><i>external</i>
"count | | ... to be defined in subclasses"<br>"empty | | ... to be defined in subclasses"<br>"put: anObject | | ... to be defined in subclasses"<br>"get | | ... to be defined in subclasses"

<i>internal</i>
<b>emptyCheck</b> | | self empty ifTrue: [↑ self error: 'queue empty'].</td></tr>
</table>

Table 14: *Class template for class* Queue.

<table>
<tr><td>class name</td><td>LinkedQueue</td></tr>
<tr><td>superclass</td><td>Queue</td></tr>
<tr><td>instance variable names</td><td>"none defined here"</td></tr>
<tr><td>class messages and methods</td><td></td></tr>
<tr><td colspan="2"><b>new</b> | | ↑ (super new) initialize.</td></tr>
<tr><td>instance messages and methods</td><td></td></tr>
<tr><td colspan="2"><i>external</i>
<b>count</b> | | ↑ buffer count.
<b>empty</b> | | ↑ buffer empty.
<b>put: anObject</b> | | buffer append: anObject.
<b>get</b> | | self emptyCheck. ↑ buffer removeAt: 1.

<i>internal</i>
<b>initialize</b> | | buffer ← CircularList new.</td></tr>
</table>

Table 15: *Class template for class* LinkedQueue.

the result. A simple example of the use of LinkedQueue is the following:

```
| queue a b c |
queue ← LinkedQueue new.
a ← (Card suit: 'heart' rank: 5).
b ← (Card suit: 'heart' rank: 6).
c ← (Card suit: 'heart' rank: 7).
queue put: a.
queue put: b.
queue put: c.
a ← queue get.
b ← queue get.
c ← queue get.
```

This sequence creates an instance of class LinkedQueue and assigns to the variables a, b, and c, instances of class Card with ranks 5, 6, and 7, respectively. These instances are put into the queue in the order listed and are then removed and assigned to the variables a, b, and c. The original order is preserved; the ranks of a, b, and c are 5, 6, and 7, respectively.

The Sequential Queue

The next implementation of a queue that we might expect to see is one that uses a sequential list to store data items. Unfortunately, a sequential list is not well suited to this purpose, because we need to add items to one end of the list and remove them from the other. You will recall that adding items to the end of a sequential list is an easy operation, but removing them from the beginning is difficult since we have to copy forward all of the succeeding items in the list. Rather than copy forward all items after the first, we would prefer to ignore the item at position 1 of the list and consider the item at position 2 to be the first item in the list. The problem, however, is that as items are added and removed from the list, the actual positions of the first and last items migrate toward the end of the list. This could cause the list to expand even if it is not full. Fortunately, we can treat the last position in the list as if it preceded the first position: that is, we consider the list to be circular. After we have added an item to the last position, we can start adding items to the beginning of the list, provided some have already been removed. If we use this strategy, then we don't have to expand the list until it is full.

The class definition in table 16 uses the strategy just described to implement a queue using sequential storage. Class SequentialQueue uses an instance of class IndexedTable as its buffer. Since class IndexedTable does not provide facilities for counting the number of objects stored in an instance (those facilities are provided by class LinearList), we need to define an instance variable count in class SequentialQueue. Additionally, we have instance variables named front and rear. front is the index of the first object stored in buffer (an instance of class IndexedTable); rear is the index of the last object stored in buffer. Since we are treating buffer as a circular sequence of positions, front and rear will repeatedly cycle through the values between the lower and upper bounds of buffer.

class name	SequentialQueue
superclass	Queue
instance variable names	front rear count
class messages and methods	

```
size: anInteger | |
   (anInteger < 1) ifTrue: [ ↑ self error: 'invalid size' ].
   ↑ (self new) buffer: (IndexedTable from: 1 to: anInteger) count: 0.
```

instance messages and methods

```
external
count | | ↑ count.
empty | | ↑ count = 0.
put: anObject | |
   (count = buffer size) ifTrue: [ self expand. ].
   buffer put: anObject at: rear.
   rear ← self advance: rear.
   count ← count + 1.
get | anObject
   | self emptyCheck.
   anObject ← buffer at: front.
   front ← self advance: front.
   count ← count − 1.
   ↑ anObject.

internal
buffer: anIndexedTable count: anInteger | |
   buffer ← anIndexedTable. count ← anInteger.
   front ← buffer lowerBound. rear ← front + count.
advance: anInteger | |
   anInteger = buffer upperBound
      ifTrue: [ ↑ buffer lowerBound ] ifFalse: [ ↑ anInteger + 1 ].
expand | anIndexedTable anInteger |
   anIndexedTable ← IndexedTable from: 1 to: (2 * buffer size).
   anInteger ← front.
   1 to: count do:
      [: i | anIndexedTable put: (buffer at: anInteger) at: i.
      anInteger ← self advance: anInteger. ].
   self buffer: anIndexedTable count: count.
```

Table 16: *Class template for class* SequentialQueue.

Because we are using an instance of class IndexedTable for storing objects, we must specify an estimate of the maximum size of an instance of class SequentialQueue when we create it. This is done with the creation message size:, which creates a new instance of SequentialQueue and sends it the internal messsage buffer:count:. The first parameter of buffer:count: is an instance of class IndexedTable; the second is the number of objects stored in the first parameter (initially zero).

The message buffer:count: is also sent from the internal message expand, which is used to expand buffer when it becomes full. expand is implemented by creating a new instance of IndexedTable that is twice as large as the current one. All of the objects stored in buffer are copied to the first half of the new instance, which then becomes the new buffer.

The internal message advance is used to advance the values of front and rear. Normally, this is done by incrementing the current value by 1. However, if the current value is equal to the upper bound of buffer, then we must set the value back to the lower bound of buffer. The external messages are those specified in the superclass Queue. The message count returns the value of the instance variable count. The message empty tests to see if count is zero.

For the message put:, we first test to see if buffer is full. If buffer is full, it is expanded using the message expand. The object passed as a parameter to put: is then stored in buffer at the position indicated by rear. rear is then advanced one position forward, using the message advance. Finally, the value of count is incremented.

Similarly, for the message get, we first test to see if the queue is empty. If it is, an error message is sent; otherwise, the object stored at the position indicated by front is removed from buffer. front is advanced one position forward, and the value of count is decremented. Finally, the removed object is returned.

Summary

The class construct is an extremely useful tool for implementing data structures. Implementing a data structure with a class makes it possible to confine the details of the implementation to one place and to insure that the resulting object will be accessed by the rest of the system in a secure manner, namely, through the use of a set of messages that correspond to the operations that are well defined for that data structure. Additionally, the ability to create subclasses makes it possible to share variables and methods among similar class definitions, thereby reducing the amount of work needed to implement a set of data structures. ■

References

1. Horowitz, E and Sartaj, S. *Fundamentals of Data Structures*. Potomac MD: Computer Science Press, 1976.
2. Knuth, D E. *The Art of Computer Programming: Volume 1/Fundamental Algorithms*, Second Edition. Reading MA: Addison-Wesley, 1973.
3. Robson, D and Goldberg, A. "The Smalltalk-80 System," August 1981 BYTE, page 36.
4. Wirth, N. *Algorithms + Data Structures = Programs*. Englewood Cliffs NJ: Prentice-Hall, 1976.

Design Principles Behind Smalltalk

Daniel H H Ingalls
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

The purpose of the Smalltalk project is to provide computer support for the creative spirit in everyone. Our work flows from a vision that includes a creative individual and the best computing hardware available. We have chosen to concentrate on two principal areas of research: a language of description (programming language) that serves as an interface between the models in the human mind and those in computing hardware, and a language of interaction (user interface) that matches the human communication system to that of the computer. Our work has followed a two- to four-year cycle that can be seen to parallel the scientific method:

- Build an application program within the current system (make an observation)
- Based on that experience, redesign the language (formulate a theory)
- Build a new system based on the new design (make a prediction that can be tested)

The Smalltalk-80 system marks our fifth time through this cycle. In this article, I present some of the general principles we have observed in the course of our work. While the presentation frequently touches on Smalltalk "motherhood," the principles themselves are more general and should prove useful in evaluating other systems and in guiding future work.

Just to get warmed up, I'll start with a principle that is more social than technical and that is largely responsible for the particular bias of the Smalltalk project:

Personal Mastery: *If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.*

The point here is that the human potential manifests itself in individuals. To realize this potential, we must provide a medium that can be mastered by a single individual. Any barrier that exists between the user and some part of the system will eventually be a barrier to creative expression. Any part of the system that cannot be changed or that is not sufficiently general is a likely source of impediment. If one part of the system works differently from all the rest, that part will require additional effort to control. Such an added burden may detract from the final result and will inhibit future endeavors in that area. We can thus infer a general principle of design:

Good Design: *A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.*

Language

In designing a language for use with computers, we do not have to look far to find helpful hints.

Everything we know about how people think and communicate is applicable. The mechanisms of human thought and communication have been engineered for millions of years, and we should respect them as being of sound design. Moreover, since we must work with this design for the next million years, it will save time if we make our computer models compatible with the mind, rather than the other way around.

Figure 1 illustrates the principal components in our discussion. A person is presented as having a body and a mind. The body is the site of primary experience, and, in the context of this discussion, it is the physical channel through which the universe is perceived and through which intentions are carried out. Experience is recorded and processed in the mind. Creative thought (without going into its mechanism) can be viewed as the spontaneous appearance of information in the mind. Language is the key to that information:

Purpose of Language: *To provide a framework for communication.*

The interaction between two individuals is represented in figure 1 as two arcs. The solid arc represents explicit communication: the actual words and movements uttered and perceived. The dashed arc represents implicit communication: the shared culture and experience that form the context of the explicit communication. In human interaction, much of the actual communication is achieved through reference to the shared context, and human language is built around such allusion. This is the case with computers as well.

It is no coincidence that a computer can be viewed as one of the participants in figure 1. In this case, the "body" provides for visual display of information and for sensing input from a human user. The "mind" of a computer includes the internal memory and processing elements and their contents. Figure 1 shows that several different issues are involved in the design of a computer language:

Scope: *The design of a language for using computers must deal with internal models, external media, and the interaction between these in both the human and the computer.*

This fact is responsible for the difficulty of explaining Smalltalk to people who view computer languages in a more restricted sense. Smalltalk is

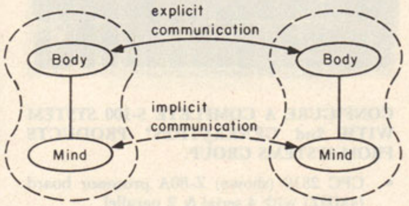


Figure 1: *The scope of language design. Communication between two people (or between one person and a computer) includes communication on two levels. Explicit communication includes the information that is transmitted in a given message. Implicit communication includes the relevant assumptions common to the two beings.*

not simply a better way of organizing procedures or a different technique for storage management. It is not just an extensible hierarchy of data types, or a graphical user interface. It is all of these things and anything else that is needed to support the interactions shown in figure 1.

Communicating Objects

The mind observes a vast universe of experience, both immediate and recorded. One can derive a sense of oneness with the universe simply by letting this experience be, just as it is. However, if one wishes to participate, literally to *take a part*, in the universe, one must draw distinctions. In so doing one identifies an object in the universe, and simultaneously all the rest becomes not-that-object. Distinction by itself is a start, but the process of distinguishing does not get any easier. Every time you want to talk about "that chair over there," you have to repeat the entire process of distinguishing that chair. This is where the act of reference comes in: we can associate a unique identifier with an object, and, from that time on, only the mention of that identifier is necessary to refer to the original object.

We have said that a computer system should provide models that are compatible with those in the mind. Therefore:

Objects: *A computer language should support the concept of "object" and provide a uniform means for referring to the objects in its universe.*

The Smalltalk storage manager provides an object-oriented model of memory for the entire system. Uniform reference is achieved simply by associating a unique integer with every object in the system. This uniformity is important because it means that variables in the system can take on widely differing values and yet can be implemented as simple memory cells. Objects are created when expressions are evaluated, and they can then be passed around by uniform reference, so that no provision for their storage is necessary in the procedures that manipulate them. When all references to an object have disappeared from the system, the object itself vanishes, and its storage is reclaimed. Such behavior is essential to full support of the object metaphor:

Storage Management: *To be truly "object-oriented," a computer system must provide automatic storage management.*

A way to find out if a language is working well is to see if the programs look like they are doing what they are doing. If they are sprinkled with statements that relate to the management of storage, then their internal model is not well matched to that of humans. Can you imagine having to prepare someone for each thing you tell them or having to inform them when you are through with a given topic and that it can be forgotten?

Each object in our universe has a life of its own. Similarly, the brain provides for independent processing along with the storage of each mental object. This suggests a third principle for object-oriented design:

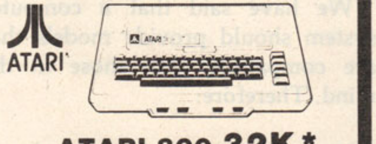
Messages: *Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.*

Just as programs get messy if object storage is dealt with explicitly, control in the system becomes complicated if processing is performed extrinsically. Let us consider the process of adding 5 to a number. In most computer systems, the compiler figures out what kind of number it is and generates code to add 5 to it. This is not good enough for an object-oriented system because the exact kind of number cannot be determined by the compiler (more on this later). A possible solution is to call a general addition routine that examines the type of the arguments to determine the appropriate action. This is not a good approach because it means that this *critical* routine must be edited by novices who just want to experiment with their own class of numbers. It is also a poor design because intimate knowledge about the internals of objects is sprinkled throughout the system.

Smalltalk provides a much cleaner solution: it sends the *name* of the desired operation, along with any arguments, as a *message* to the number, with the understanding that the receiver knows best how to carry out the desired operation. Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires. The transmission of messages is the only process that is carried on outside of objects and this is as it should be, since messages travel between objects. The principle of good design can be restated for languages:

Uniform Metaphor: *A language should be designed around a powerful metaphor that can be uniformly applied in all areas.*

Examples of success in this area include LISP, which is built on the model of linked structures; APL, which is built on the model of arrays; and Smalltalk, which is built on the model of communicating objects. In each case, large applications are viewed in the same way as the fundamental units from which the system is built. In Smalltalk especially, the interaction between the most primitive objects is viewed in the same way as the highest-level interaction between the computer and its user. Every object in Smalltalk, even a lowly integer, has a set of messages, a *protocol*, that defines the explicit communication to which that object can respond. Internally, objects may have local storage and access to other shared information which comprise the implicit context of all communication. For instance, the message + 5 (add five) carries an implicit assumption that the augend is the present



value of the number receiving the message.

Organization

A uniform metaphor provides a framework in which complex systems can be built. Several related organizational principles contribute to the successful management of complexity. To begin with:

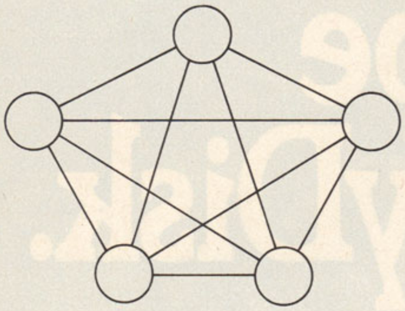


Figure 2: *System complexity. As the number of components in a system increases, the chances for unwanted interaction increase rapidly. Because of this, a computer language should be designed to minimize the possibilities of such interdependence.*

Modularity: *No component in a complex system should depend on the internal details of any other component.*

This principle is depicted in figure 2. If there are *N* components in a system, then there are roughly *N-squared* potential dependencies between them. If computer systems are ever to be of assistance in complex human tasks, they must be designed to minimize such interdependence. The message-sending metaphor provides modularity by decoupling the *intent* of a message (embodied in its name) from the *method* used by the recipient to carry out the intent. Structural information is similarly protected because all access to the internal state of an object is through this same message interface.

The complexity of a system can often be reduced by grouping similar components. Such grouping is achieved through data typing in conventional programming languages, and through *classes* in Smalltalk. A class describes other objects—their internal state, the message protocol they recognize, and the internal methods for responding to those messages. The objects so described are called *instances* of that class. Even classes themselves fit into this framework; they are just instances of class Class, which describes the appropriate protocol and implementation for object description:

Classification: *A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the kernel classes of the system.*

Classification is the objectification of *ness*ness. In other words, when a human sees a chair, the experience is taken both literally as "that very thing" and abstractly as "that chair-like thing." Such abstraction results from the marvelous ability of the mind to merge "similar" experience, and this abstraction manifests itself as another object in the mind, the Platonic chair or chair*ness*.

Classes are the chief mechanism for extension in Smalltalk. For instance, a music system would be created by adding new classes that describe the representation and interaction protocol of Note, Melody, Score, Timbre, Player, and so on. The "equal footing" clause of the above principle is important because it insures that the system will be used as it was designed. In other words, a melody could be represented as an ad hoc collection of Integers representing pitch, duration, and other parameters, but if the language can handle Notes as easily as Integers, then the user will naturally describe a melody as a collection of Notes. At each stage of design, a human will naturally choose the most effective representation if the system provides for it. The principle of modularity has an interesting implication for the procedural components in a system:

Polymorphism: *A program should specify only the behavior of objects, not their representation.*

A conventional statement of this principle is that a program should never declare that a given object is a SmallInteger or a LargeInteger, but only that it responds to integer protocol. Such generic description is crucial to models of the real world.

Consider an automobile traffic simulation. Many procedures in such a system will refer to the various vehicles involved. Suppose one wished to add, say, a street sweeper. Substantial amounts of computation (in the form of recompiling) and possible errors would be involved in making this simple extension if the code depended on the objects it manipulates. The message interface establishes an ideal framework for such extension. Provided that street sweepers support the same protocol as all other vehicles, no changes are needed to include them in the simulation:

Factoring: *Each independent component in a system should appear in only one place.*

There are many reasons for this principle. First of all, it saves time, effort, and space if additions to the system need only be made in one place. Second, users can more easily locate a component that satisfies a given need. Third, in the absence of proper factoring, problems arise in synchronizing changes and ensuring that all interdependent components are consistent. You can see that a failure in factoring amounts to a violation of modularity.

Smalltalk encourages well-factored designs through *inheritance*. Every class inherits behavior from its superclass. This inheritance extends through increasingly general classes, ultimately ending with class Object which describes the default behavior of all objects in the system. In our traffic simulation above, StreetSweeper (and all other vehicle classes) would be described as a subclass of a general Vehicle class, thus inheriting appropriate default behavior and avoiding repetition of the same concepts in many different places. Inheritance illustrates a further pragmatic benefit of factoring:

Leverage: *When a system is well factored, great leverage is available to users and implementers alike.*

Take the case of sorting an ordered collection of objects. In Smalltalk, the user would define a message called sort in the class OrderedCollection. When this has been done, all forms of ordered collections in the system will instantly acquire this new capability through inheritance. As an aside, it is worth noting that the same method can alphabetize text as well as sort numbers, since comparison protocol is recognized by the classes which support both text and numbers.

The benefits of structure for implementers are obvious. To begin with, there will be fewer primitives to implement. For instance, all graphics in Smalltalk are performed with a single primitive operation. With only one task to do, an implementer can bestow loving attention on every instruction, knowing that each small improvement in efficiency will be amplified throughout the system. It is natural to ask what set of primitive operations would be sufficient to support an entire computing system. The answer to this question is called a *virtual machine* specification:

Virtual Machine: *A virtual machine specification establishes a framework for the application of technology.*

The Smalltalk virtual machine establishes an object-oriented model for storage, a message-oriented model for processing, and a bitmap model for visual display of information. Through the use of microcode, and ultimately hardware, system performance can be improved dramatically without any compromise to the other virtues of the system.

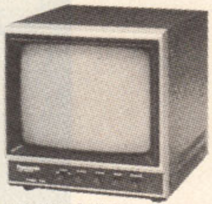
User Interface

A user interface is simply a language in which most of the communication is visual. Because visual presentation overlaps heavily with established human culture, esthetics plays a very important role in this area. Since all capability of a computer system is ultimately delivered through the user interface, flexibility is also essential here. An enabling condition for adequate flexibility of a user interface can be stated as an object-oriented principle:

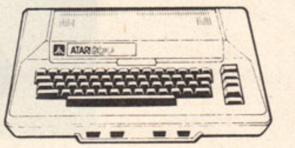
Reactive Principle: *Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.*

This criterion is well supported by the model of communicating objects. By definition, each object provides an appropriate message protocol for interaction. This protocol is essentially a microlanguage particular to just that kind of object. At the level of the user interface, the appropriate language for each object on the screen is presented visually (as text, menus, pictures) and sensed through keyboard activity and the use of a pointing device.

It should be noted that operating systems seem to violate this principle. Here the programmer has to depart



TR-930 9" Desk-Top Monitor



from an otherwise consistent framework of description, leave whatever context has been built up, and deal with an entirely different and usually very primitive environment. This need not be so:

Operating System: *An operating system is a collection of things that don't fit into a language. There shouldn't be one.*

Here are some examples of conventional operating system components that have been naturally incorporated into the Smalltalk language:

- Storage management—Entirely automatic. Objects are created by a message to their class and reclaimed when no further references to them exist. Expansion of the address space through virtual memory is similarly transparent.
- File system—Included in the normal framework through objects such as Files and Directories with message protocols that support file access.
- Display handling—The display is simply an instance of class Form, which is continually visible, and the graphical manipulation messages defined in that class are used to change the visible image.
- Keyboard input—The user input devices are similarly modeled as objects with appropriate messages for determining their state or reading their history as a sequence of events.
- Access to subsystems—Subsystems are naturally incorporated as independent objects within Smalltalk: there they can draw on the large existing universe of description, and those that involve interaction with the user can participate as components in the user interface.
- Debugger—The state of the Smalltalk processor is accessible as an instance of class Process that owns a chain of stack frames. The debugger is just a Smalltalk subsystem that has access to manipulate the state of a suspended process. It should be noted that nearly the only run-time error that can occur in Smalltalk is for a message not to be recognized by its receiver.

Smalltalk has no "operating system" as such. The necessary primitive operations, such as reading a page from the disk, are incorporated as primitive methods in response to otherwise normal Smalltalk messages.

Future Work

As might be expected, work remains to be done on Smalltalk. The easiest part to describe is the continued application of the principles in this paper. For example, the Smalltalk-80 system falls short in its factoring because it supports only hierarchical inheritance. Future Smalltalk systems will generalize this model to arbitrary (multiple) inheritance. Also, message protocols have not been formalized. The organization provides for protocols, but it is currently only a matter of style for protocols to be consistent from one class to another. This can be remedied easily by providing proper protocol objects that can be consistently shared. This will then allow formal typing of variables by protocol without losing the advantages of polymorphism.

The other remaining work is less easy to articulate. There are clearly other aspects to human thought that have not been addressed in this paper. These must be identified as metaphors that can complement the existing models of the language.

Sometimes the advance of computer systems seems depressingly slow. We forget that steam engines were high-tech to our grandparents. I am optimistic about the situation. Computer systems are, in fact, getting simpler and, as a result, more usable. I would like to close with a general principle which governs this process:

Natural Selection: *Languages and systems that are of sound design will persist, to be supplanted only by better ones.*

Even as the clock ticks, better and better computer support for the creative spirit is evolving. Help is on the way.■

The Smalltalk-80 Virtual Machine

Glenn Krasner
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

The Smalltalk-80 system is a powerful system that encourages the development of large applications programs. The system contains a compiler, a debugger, a storage management system, text and picture editors, and a file system. It also contains a highly interactive user interface based on graphics that include overlapping windows.

Typically the task of bringing up such a powerful system on a new computer includes writing code to implement these pieces. The Smalltalk-80 system is different in that most of these pieces are written in Smalltalk-80 itself. The part that can be written in Smalltalk-80 is called the *Smalltalk-80 Virtual Image*, and it includes the compiler, debugger, editors, decompiler, and the file system.

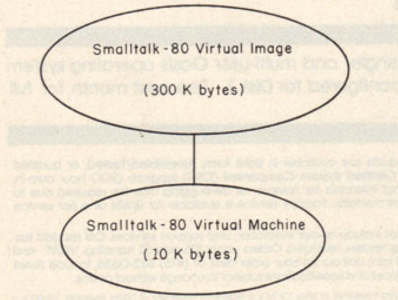


Figure 1: *The Smalltalk-80 Virtual Machine. Most of Smalltalk-80 is written in Smalltalk-80 (the Virtual Image), leaving only a small amount of code that has to be rewritten for each processor on which the language is implemented (the Virtual Machine).*

The remaining part of the Smalltalk-80 system is defined in terms of an abstract machine called the *Smalltalk-80 Virtual Machine* (see figure 1). The Smalltalk-80 compiler translates source code into machine instructions for this virtual machine, rather than translating directly into machine instructions for a particular hardware machine. The task of bringing up a Smalltalk-80 system on a new "target" computer consists only of implementing (writing a program to simulate) the Smalltalk Virtual Machine on the target computer.

In this article, we will present an overview of the elements needed to implement the Smalltalk Virtual Machine. These elements are:

- the *Storage Manager*
- the *Interpreter*
- the *Primitive Subroutines*

Background

A Smalltalk-80 system is made up of *objects* that have state and exhibit behavior. Their state consists of the values of both named and indexed instance variables (which we will call *fields*), and their behavior is exhibited through sending and receiving *messages*. Objects are members of *classes*.

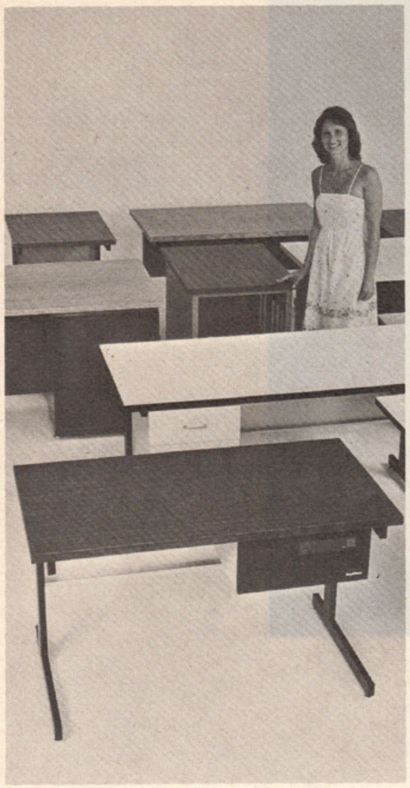
Classes may be *subclasses* of other classes—that is, they may inherit attributes from other classes. Programming in Smalltalk-80 is done by defining the procedures, or *methods*, that are executed when objects receive messages. Typically, messages are sent to other objects to invoke their methods. Sometimes messages invoke *primitive* (machine-code) *subroutines* rather than Smalltalk-80 methods.

From this brief description of Smalltalk-80, we can consider the information needed to implement each of the three elements of the Smalltalk Virtual Machine:

1. To implement the storage manager, we need the information necessary to represent objects in the computer's memory. This information consists of the amount of memory that each object will occupy, which can be computed from the number of fields the object has, and the representation of fields in memory. Objects that describe classes define the number of fields their instances will have, so we also need to know how this number is represented. With this information, we can design a storage manager for objects in a Smalltalk-80 system that will:

- fetch the class of objects
- fetch and store fields of objects
- create new objects
- collect and manage free space

2. The interpreter executes the machine instructions of the Smalltalk-80 Virtual Machine. The information needed to design the interpreter is a description of these machine instructions, called bytecodes (the idea is similar to Pascal p-codes). The bytecodes are contained in methods, so we also need to know the representa-



Who's Who

The design of the Smalltalk-80 Virtual Machine is based on previous Smalltalk systems implemented by the Learning Research Group at Xerox PARC. The original bytecode interpreter design was made for Smalltalk-76 by Dan Ingalls (Ingalls, Dan. "The Smalltalk-76 Programming System: Design and Implementation." In Fifth Annual ACM Symposium on Principles of Programming Languages, *1978, pages 9 through 16). Smalltalk-76 was implemented on the Xerox Alto by Dan Ingalls, Ted Kaehler, Dave Robson, Steve Weyer and Diana Merry, on the Xerox Dolphin by Peter Deutsch, and on the Xerox Dorado by Bruce Horn. TinyTalk was implemented on a Xerox microcomputer by Larry Tesler and Kim McCall (McCall, Kim and Larry Tesler. "Tiny Talk, a Subset of Smalltalk-76 for 64KB Microcomputers." In* Proceedings of the Third Symposium on Small Systems, *ACM Sigsmall Newsletter, Volume 6, Number 2, 1980, pages 197 through 198). Smalltalk-78 (a revised version of Smalltalk-76 similar to Smalltalk-80) was implemented on the Xerox microcomputer by Dan Ingalls, Ted Kaehler, and Bruce Horn, on the Xerox Dorado by Jim Stamos, and on a Norwegian microcomputer (under a research license from Xerox) by Bruce Horn. Smalltalk-80 has been implemented on the Xerox Dorado by Peter Deutsch, on the Xerox Dolphin by Kim McCall, and on the Xerox Alto by Glenn Krasner. The designs of these systems were made by the implementors and other members of the Learning Research Group.*

tion of methods. From this information we can decide how the interpreter will fetch and execute bytecodes and how it will find methods to run when messages are sent.

3. The last piece of information we need to know is which messages will invoke primitive subroutines; that is, which methods we must implement in machine code to terminate the recursion of message sending and to optimize performance.

Before we go into more detail about these elements of a Smalltalk-80 Virtual Machine implementation, here are a few typical figures that will provide a little "reality" to implementors. For the systems that we have implemented at Xerox, the Smalltalk-80 Virtual Image consists of about 300 K bytes of objects. Our typical implementation of the Smalltalk-80 Virtual Machine is 6 to 12 K bytes of assembly code, or 2 K microcode instructions plus 10 K bytes of assembly code. Of this, about 40% is in the storage manager, 20% in the interpreter, and 40% in the primitive subroutines. Our average is about one person-year to implement a fully debugged version of this code.

The Storage Manager

Although the storage manager tends to be the largest and most complex of the three parts of a Smalltalk-80 implementation, the functions it provides are few and relatively simple to understand.

Everything in a Smalltalk-80 system is an object.

Everything in a Smalltalk system is an object, so from a storage point of view memory needs to be divided into blocks, one for each object, plus a pool of memory that is not yet used. Every time a new object is created, a new block of the appropriate size must be found for that object: when objects are no longer used, their memory block may be returned to the pool (see figure 2).

A special entity called an *object pointer* is assigned to each object. If an object pointer were the actual core address of the memory occupied by that object, then there would be fast access to an object given its pointer. However, in the Smalltalk-80 system the object pointer is an *indirect* pointer to the object through a table kept by the storage manager. This allows the storage manager to move an object around in memory without affecting any object that refers to it. It also insures that the storage manager is the only entity in the system concerned with (and allowed to change)

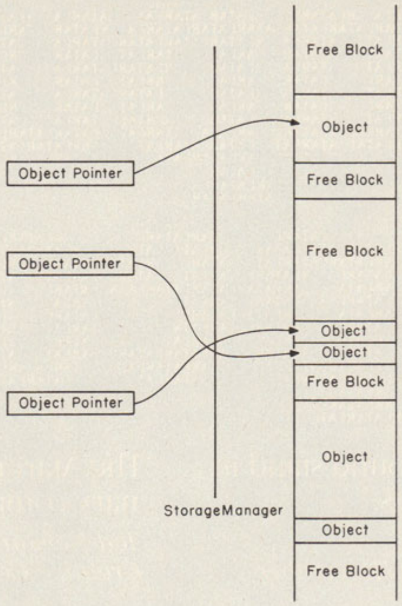


Figure 2: *Objects and memory usage in Smalltalk-80. Each Smalltalk-80 object has an object pointer that points to a block of memory that describes the object. When an object is no longer used, its memory is made available for use.*

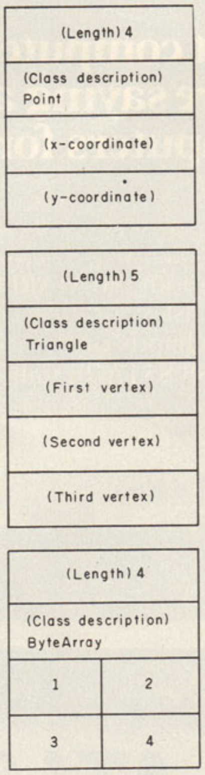


Figure 3: *Typical object representations in Smalltalk-80.*

the actual memory. In the Smalltalk-80 Virtual Image, object pointers are single 16-bit words. This allows for 64 K objects in the system; these objects may take up much more than 64 K words of memory.

Since an object's class and fields are themselves objects, we can see that the block of memory corresponding to an object contains the object pointer of the object's class plus the object pointer for each of the object's fields. The storage manager also keeps the length of the block as one word of the block. This means, for example, that the block corresponding to an object that is an instance of class Point (see figure 3) will have:

- one word that says this block is four words long
- one word that is the object pointer of the object that describes class Point
- one word that is the object pointer of an object that is the *x*-coordinate field of the point
- one word that is the object pointer of an object that is the *y*-coordinate field of the point

Similarly, the block corresponding to an object that is an instance of class Triangle will have:

- one word saying this block is five words long
- one word that is the object pointer of the object that describes class Triangle
- one word that is the object pointer of an instance of class Point, representing one vertex field
- one word that is the object pointer of an instance of class Point, for the second vertex field
- one word that is the object pointer of an instance of class Point, for the third vertex field

For performance optimization, the values in the fields of some objects, such as instances of class ByteArray, will be interpreted as the numerical values themselves, rather than as object pointers. The block corresponding to the byte array containing the elements 1, 2, 3, and 4, in order, will have:

- one word saying this block is four words long
- one word pointing to the object that describes class ByteArray
- one byte encoding the number 1
- one byte encoding the number 2
- one byte encoding the number 3
- one byte encoding the number 4

We will represent all objects as having fields interpreted as object pointers or numerical values, not both. Objects may store numerical values as bytes or words, but not both.

As we have mentioned, the objects that describe classes also need to represent the form of instances of those classes. The essential information is the number of fields the instances will have, and whether these will be pointer or nonpointer fields. For example, the describer of class Point says that its instances will have two fields (*x*- and *y*-coordinates) and that these will be pointers (see figure 4). The describer of class ByteArray says that its instances may have a variable number of fields and that these fields will not be pointers but will be numerical values stored in bytes.

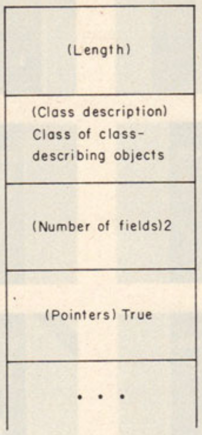


Figure 4: *Class-describing object for class* Point.

The purpose of the storage manager is to fetch and store fields of objects, to create objects, and to manage free space. A clean implementation of the storage manager would be one in which the other parts of the system had access only to the object pointers and made requests of the storage manager only through the following subroutine calls:

- getClass(objectPointer) returns the object pointer of the class of the given object
- getField(objectPointer,fieldOffset) returns the field
- storeField(objectPointer,fieldOffset,newValue) replaces that field with the new value newValue
- newInstance(classObjectPointer,numberOfFields) returns the object pointer of a new instance of that class, and, if that class can have indexed instance variables, this instance has the given number of fields (numberOfFields)

Requests can be made for new storage (with the newInstance subroutine), but not to return used storage. In some other systems, storage that is no longer used must be explicitly returned to the free storage pool. The Smalltalk-80 philosophy is that neither the user nor any part of the system other than the storage manager need have such concerns. Therefore the storage manager must know which objects are no longer being used, so that their storage may reenter the free pool. Typically, Smalltalk-80 Virtual Machine implementations use *reference-counting* to accomplish this. For every object in the system, the storage manager keeps a count of the number of other objects that point to it. This number will change only during execution of the four storage-manager subroutines. When this count reaches zero, the object's memory block may be reused because there are no references to that object anywhere else in the system.

The Interpreter

The interpreter is that portion of the Smalltalk-80 Virtual Machine that performs the actions described in the bytecodes of methods (ie: the machine code of the Virtual Machine). The information needed to implement the interpreter is the

description of the bytecodes, the representation of methods, and the technique to find the method to run when sending a message.

The bytecodes define the Smalltalk-80 Virtual Machine as a stack-oriented machine. Each bytecode represents one of the following actions:

- push an object onto the stack
- store the top of the stack as the value for a variable
- pop the top of the stack
- branch to another bytecode
- send a message using the top few elements of the stack
- return the top of the stack as the value for this method

In the Smalltalk-80 Virtual Machine, each of these actions is realized by one or more bytecodes. Note that pushing, storing, popping, and branching are standard instruction types for any stack machine, that sending a message corresponds to calling a procedure using the top few elements of the stack as arguments, and that returning an object from a method corresponds to returning a value from a procedure. The difference between the Smalltalk-80 Virtual Machine and procedure-based stack machines is in the way the procedure is found. In most procedure-based stack machines the address of a procedure is provided in the *execute procedure* instruction; in the Smalltalk-80 system only the "name," called the *selector*, of the message is provided; the method (or procedure) to be executed is found through a strategy involving the receiver of the message and its class. We will first describe the bytecodes, then how methods are represented, and finally give a strategy for finding methods.

Stack Operations

The Smalltalk-80 Virtual Machine and corresponding bytecode set are stack oriented. Object pointers are pushed and popped from a stack, and when a message is sent, the top few elements of the stack are used as receiver and arguments of the method. These are replaced by the object returned as the value of that method. For example, the Smalltalk-80 expression:

3 * (4 + 5)

is encoded by the bytecodes shown in table 1.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Push 4	(3 4)
-3- Push 5	(3 4 5)
-4- Send +	(3 9)
-5- Send *	(27)

Table 1: *Bytecodes for the Smalltalk expression* 3 * (4 + 5).

As bytecodes labeled -1-, -2-, and -3- are executed by the interpreter, the objects 3, 4, and 5 are pushed onto the stack. When bytecode -4- is executed, the message + is sent to the second object on the stack (4) with the top object of the stack as the argument (5). The 4 and 5 are popped off this stack when the message is sent, and the interpreter begins executing the bytecodes for the method corresponding to the message + in the Smalltalk class of small integers. This method will eventually return an object, in this case 9, as its value, and the interpreter will push the 9 onto the original stack above the 3 and resume execution with bytecode -5-. Bytecode -5- will produce an effect similar to that produced by -4-, leaving the object 27 on the stack. In the same way that other stack machines push *data* onto a *stack* and use the top few data items as *arguments* for a *procedure*, replacing them with the *value* returned from that procedure, the Smalltalk-80 Virtual Machine pushes *object pointers* onto a *stack*

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Push 4	(3 4)
-3- Send +	(7)
-4- Store into a	(7)

Table 2: *Bytecodes for the Smalltalk expression* a ← 3 + 4.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Store into a	(3)
-3- Pop	( )
-4- Push 4	(4)
-5- Store into b	(4)

Table 3: *Bytecodes for the Smalltalk expression* a ← 3. b ← 4.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Store into a	(3)
-3- Pop	( )
-4- Push a	(3)
-5- Return top of stack	( )

Table 4: *Bytecodes for the Smalltalk expression* a ← 3. ↑ a.

and uses the top few as *receiver* and *arguments* of a *message*, replacing them with the *object* returned from that method.

In both machines, values from the top of the stack may be stored as the values of variables. As an example, the Smalltalk expression:

```
a ← 3 + 4
```

will be represented by the bytecodes in table 2. Here, -1-, -2- and -3- act as before and the interpreter executes bytecode -4- by storing the top of the stack 7 into the variable a.

Stack machines in general, and the Smalltalk-80 Virtual Machine in particular, also have the ability to pop the top element off the stack. In the statements:

```
a ← 3.
b ← 4
```

once the 3 is stored into variable a, it is no longer needed, so it is popped from the stack. These statements are represented by the bytecodes shown in table 3.

The top of the stack may be returned as the value for the method. The statements:

```
a ← 3.
↑ a
```

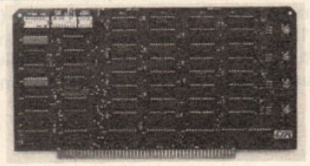
are represented by the bytecodes shown in table 4.

Branching Operations

Conditional and looping messages are used so often that they are represented not by actual messages but by bytecodes for conditional and unconditional jumps. (This is *only* for performance reasons; these branching and looping messages would work if they were actually sent like other messages.) For example:

```
a > 4 ifTrue: [a ← a − 1]
```

(which in the Smalltalk-80 system means execute the code within the brackets only if the object returned from the > message is not false) is represented in table 5 (ignoring the stack from now on).



Bytecode

```
-1-  Push 4
-2-  Push a
-3-  Send >
-4-  Jump to -10- if the top of the stack is false
-5-  Push a
-6-  Push 1
-7-  Send —
-8-  Store into a
-9-  Pop
-10- < the next bytecode >
```

Table 5: *Bytecodes for the Smalltalk expression* a > 4 ifTrue: [a ← a − 1].

Bytecode

```
-1-  Push a
-2-  Push 4
-3-  Send >
-4-  Jump to -11- if top of stack is false
-5-  Push a
-6-  Push 1
-7-  Send —
-8-  Store into a
-9-  Pop
-10- jump to -1-
-11- < the next bytecode >.
```

Table 6: *Bytecodes for the Smalltalk expression* [a > 4] whileTrue: [a ← a − 1].

Table 6 shows the bytecodes for the looping expression:

[a > 4] whileTrue: [a ← a − 1]

(which means execute the code in the second brackets as long as the code in the first set of brackets evaluates to something other than false).

Addressing Variables

Methods are implemented as objects whose fields contain the bytecodes plus a group of pointers to other objects called the *literal frame*. The interpreter can use the getField subroutine of the storage manager to fetch the next required bytecode to execute. This takes care of returns, jumps, and pops, but for the other bytecodes we need to represent more information. In particular, for the push and store bytecodes, we need to represent where to find the object pointers to push; for the send bytecodes, we need to represent where to find the selector of the message and which stack elements are the receiver and arguments.

The source code for a method contains variable names and literals, but the bytecodes of the Virtual Machine are defined only in terms of field offsets. From the Virtual Machine's point of view, there are three types of variables: variables local to the method (called *temporaries*), variables local to the receiver of the message (*instance variables*), or variables found in some dictionary that the receiver's class shares (*global variables*). Note that *class variables* are treated in the same way as other global variables. The Smalltalk-80 compiler (itself written in Smalltalk-80) translates references to these variables into bytecodes that are references to field offsets of the receiver, the temporary area, or globals. The instance variables are translated using a field of class-describing objects that associates instance variable names with field offsets. The assignment of offsets to temporaries is done when the compiler translates a method by associating names of temporaries to offsets in the temporary area. The compiler creates instances for the literals, puts their object pointers into the literal frame of the method, and produces bytecodes in terms of offsets into the literal frame. For global variables, the compiler uses system dictionaries that associate global names to indirect references to objects. Object pointers of the indirect references to the global objects are also placed in the literal frame of the method. The bytecodes for accessing globals are encoded as indirect references through field offsets of the literal frame.

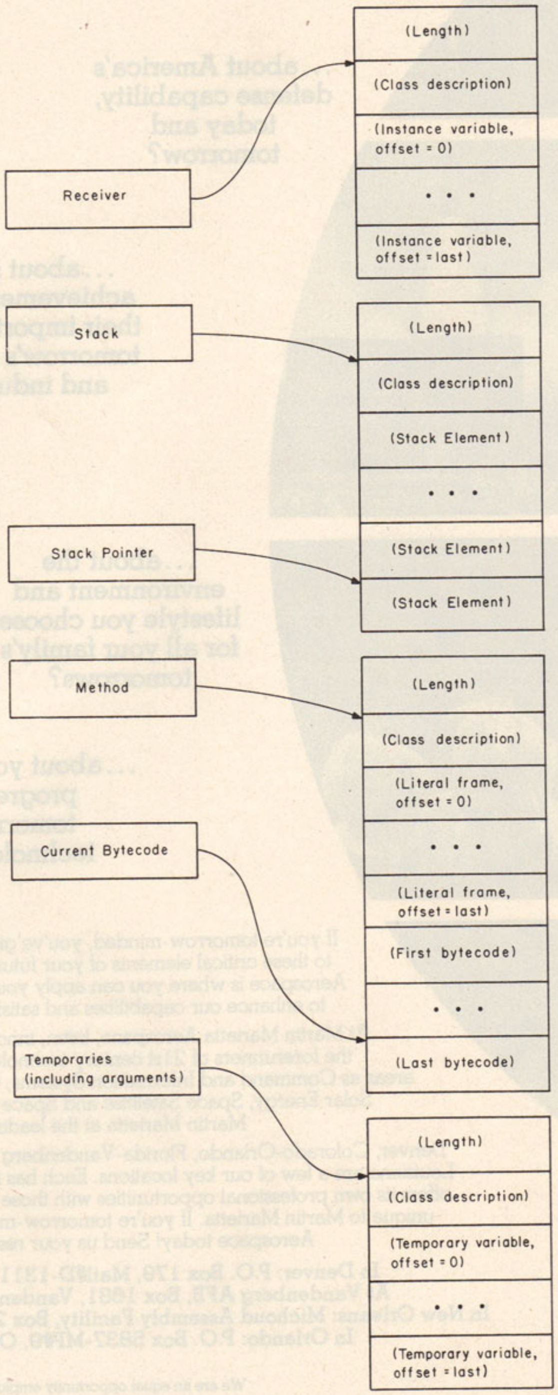


Figure 5: *Object pointers held by the interpreter.*

This means that when the interpreter is executing a method, it has to keep a stack, a temporary area, a pointer to the receiver and arguments of the method, and a pointer to the method itself (see figure 5). It uses the storage manager's getField and storeField subroutines to push and pop pointers from the stack object, to retrieve and set values of variables in the temporary area, to retrieve and set values of variables of the receiver, and to get bytecodes and values of global variables from the method.

Finding Methods

When a message is sent, the receiver and arguments must be identified, and the appropriate method must be found by the interpreter. The technique used in Smalltalk-80 is to include in each class-describing object a dictionary, called the *method dictionary*, that associates selectors with methods. Pointers to the selectors that will be sent by any method are kept in the method (along with global variable pointers and bytecodes). The bytecodes that tell the interpreter to send a message encode a field offset in the literal frame where the selector is found, plus the number of arguments that that method needs. By convention, the top elements of the stack are the arguments and the next one down is the receiver. For example, the send bytecode for the expression:

```
3 + 4
```

will stand for "send the selector in field X of the method (which will be +), and it takes one argument." The interpreter will ask the storage manager for the X field of the method, will get the top of the stack (4) as the argument, and the next element down (3) as the receiver. It will locate the receiver's class, its method dictionary, search it for an association of the + selector with some method, and, when found, execute that method.

If no such association is found, the searching does not end. The receiver's class may be a subclass of another class, called its *superclass*. If this is the case, the method for + may be

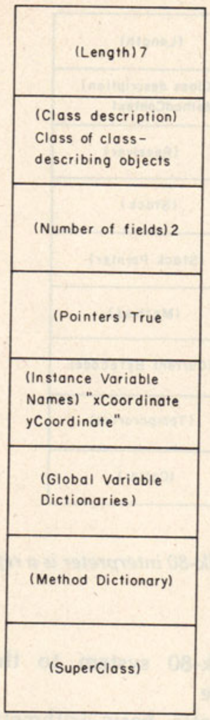

(Length) 7
(Class description) Class of class-describing objects
(Number of fields) 2
(Pointers) True
(Instance Variable Names) "xCoordinate yCoordinate"
(Global Variable Dictionaries)
(Method Dictionary)
(SuperClass)

Figure 6: *Class-describing object for class* Point, *revisited.*

defined in the superclass, so the interpreter must check there. This means that each class must have a field that refers to its superclass (see figure 6). The interpreter searches the method dictionary of the superclass, its superclass, and so on, until either an appropriate method is found or it runs out of superclasses, in which case an error occurs.

To execute a method, the interpreter needs a place for temporaries and a stack for that method. In the Smalltalk-80 Virtual Machine, this is done by allocating an object that is an instance of class MethodContext. Objects in MethodContext keep track of the method, the stack for that method, a pointer to the next bytecode to be executed in that method, the temporary variables for that method, and the context from which that method was invoked, called the *caller* of that method (see figure 7). When a method returns, the value returned is pushed on the stack of the caller context, and execution continues at the next bytecode of the caller's method.

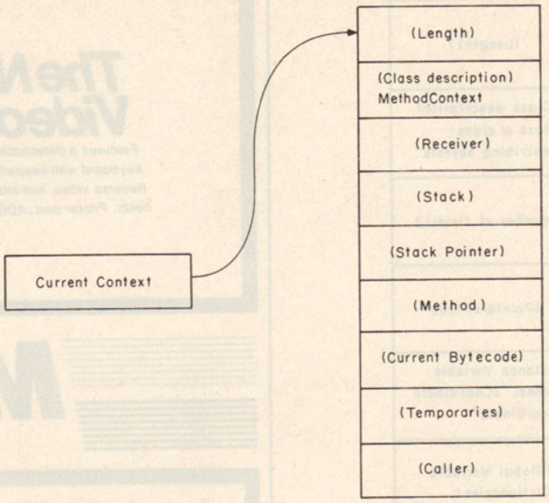


Figure 7: *The only object pointer used by the Smalltalk-80 interpreter is a reference to a* MethodContext.

The Smalltalk-80 Virtual Machine implementation is a program running in the machine language of the target computer.

Primitive Subroutines

The Smalltalk-80 Virtual Machine implementation is a program running in the machine language of the target computer. The storage manager is the collection of subroutines in this program that deals with memory allocation and deallocation. The interpreter is the collection of subroutines in this program, one of which fetches the next bytecode from the currently running method and calls one of the others to perform the appropriate action for that bytecode. In addition to these functions, we have found that there are several other places in the Smalltalk-80 system where performance considerations make it necessary, or at least desirable, to implement certain functions as machine-code subroutines in the Smalltalk-80 Virtual Machine. These places are:

- input/output: connecting the Smalltalk-80 system to the actual hardware
- arithmetic: basic arithmetic for integers
- subscripting indexable objects: fetching and storing indexable instance variables
- screen graphics: drawing and moving areas of the screen bitmap quickly
- object allocation: connecting the Smalltalk-80 code for creating a new instance with the storage manager subroutines

We call this set of subroutines the *primitive subroutines*.

The primitive subroutines are represented in the Smalltalk Virtual Image as methods with a special flag that says to run the corresponding subroutine rather than the Smalltalk-80 bytecodes. When the interpreter is executing the code to send a message and finds one of these flags set, it calls the subroutine and uses the value returned from it as the value of the method. The number of these methods in Smalltalk-80 is small (around one hundred) in order to keep the rest of the system as flexible and extensible as possible. We will not list those methods that are primitives, but will refer the reader to *Smalltalk: the Language and Its Implementation* (Goldberg, Robson, and Ingalls, 1981) for details.

A few of these primitive methods are executed so often that even the cost of looking them up in their classes' method dictionaries would be excessive. These methods are instead represented as special versions of the Send Message type of bytecodes. The message + , for example, is represented this way. When this bytecode is executed and the top two elements of the stack are small integers, then the primitive method is called as a subroutine. When this bytecode is executed and the top two elements of the stack are not small integers, then the + message is sent normally.

Conclusion

The Smalltalk-80 Virtual Machine is a fairly small computer program that consists of a storage manager, an interpreter, and a set of primitive subroutines. The task of implementing a Smalltalk-80 Virtual Machine for a new target computer is not large (especially when compared with the task of implementing other large programming systems) because most of the functions that must usually be implemented in machine code are already part of the Smalltalk-80 Virtual Image that runs on top of the Virtual Machine.

The Smalltalk-80 Virtual Machine could also be implemented in hardware, although this has not yet been done. Such an implementation would sacrifice some of the flexibility of software, but it would result in the performance benefits that hardware provides. Given the evolving nature of Smalltalk, it may not yet be time to implement the Virtual Machine in hardware: new Smalltalks that are more powerful would likely need at least small changes in Virtual Machine definition and implementation. However, hardware assists to Smalltalk-80 Virtual Machine software can greatly improve performance. Writable microcode stores for the pieces of code that are frequently run, hardware assists for graphics, or hardware assists for the fetching of bytecodes could all potentially improve the performance of a Smalltalk-80 Virtual Machine implementation. ■

Building Control Structures in the Smalltalk-80 System

L Peter Deutsch
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Just as *data structures* refer to the ways that we group data together by using simple *objects* to represent more complex objects, *control structures* refer to the ways a programmer can build up complex sequences of *operations* from simpler ones. The easiest example of a control structure is sequencing: do something and then do something else. Two other familiar examples are the *conditional* structure (if some condition is true, do something, otherwise do something else) and the *loop* (do something as long as some condition remains true).

Most languages provide a few common control structures, typically sequencing, conditional, looping, and procedures, but no way for a programmer to define new structures. One useful control structure that many languages omit is the simple *case* statement (given *N* alternative things to do, numbered from 1 to *N*, and a variable *K*, do the *K*th thing). If the language doesn't provide a case statement, you can always simulate it with a long string of conditionals, but it makes your program harder to read. Other useful control structures are much more difficult to simulate if the language fails to provide them.

The Smalltalk-80 language and system (which will be called simply "Smalltalk") is one of the few languages in which a programmer can invent and implement, with relative ease, new control structures that aren't provided by the system implementors. The rest of the article illustrates this point with examples that have actually been run on a Smalltalk-80 implementation.

What's Built In

Smalltalk provides very few built-in control structures. There is the conditional structure, implemented as follows:

```
someCondition ifTrue: [somethingToDo]
someCondition ifFalse: [somethingToDo]
someCondition
   ifTrue: [somethingToDo]
   ifFalse: [somethingElseToDo]
```

and the simple loop:

```
[someCondition] whileTrue: [somethingToDo]
[someCondition] whileFalse: [somethingToDo]
```

The most powerful tool for building new structures is the *block*. Two examples are:

```
[somethingToDoLater]
```

and:

```
[:anArgumentName| somethingToDoLater]
```

The block allows a caller to pass to the implementor of a control structure a piece of code to be executed (possibly with arguments, as in the second example) at an appropriate time.

Case Statement

Our first example is the case statement described before. We would like a construct that includes an indexed collection of blocks for the expected cases and another block for the situation where the index is out of range. Without any particular trouble we can have a construct like this:

```
someExpression
   case: (Array with: [case1] with: [case2] with:
         [case3])
   otherwise: [somethingElse]
```

where [somethingElse] gets evaluated if someExpression isn't 1, 2, or 3. Then the definition is simple. We add a message to the existing class Number. In order to distinguish adding methods to existing classes from creating new classes, we will label templates "existing" if they are to be seen as partial templates adding new methods to existing classes.

Table 1 shows the code necessary to add the case method to the class Number. As far as the control structure goes, this is all there is to it: alternativeBlocks is an array of blocks, and the method in Number simply picks the appropriate one to evaluate. However, the syntax looks clumsy. We might like to have something that looks more like the following:

```
someExpression
    case: [case1], [case2], [case3]
    otherwise: [somethingElse]
```

One way to do this is to arrange for appropriate interpretation of the message , (comma) by BlockContext (table 2a) and to invent a new subclass of IndexedCollection (table 2b) that will also interpret , appropriately. We also have to add some protocol to BlockContext to handle the situation of a single block. Note that double quote marks delineate comments in Smalltalk.

As a matter of style, we generally discourage syntactic embellishments of this kind: their implementation tends to be obscure and they don't add that much to the ease of writing programs.

Generator Loops

Many languages provide a kind of loop called a *generator*, which sets a variable to successive values generated by some algorithm each time through the interior of the loop. The familiar kind of loop that runs through successive integers from 1 to *N* is one such example. Another example is looking at successive elements of a linked list, or any ordered collection.

Smalltalk actually provides simple generators of the form:

```
someCollection do:
    [:anElement| doSomethingWithTheElement]
```

but it is instructive to see how we could have constructed them ourselves. This could be accomplished by having each kind of collection object implement the message do:

class name (existing)	Number
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
    control
    case: alternativeBlocks otherwise: aBlock | |
        (self >= 1 and: [self <= alternativeBlocks size])
            ifTrue: [↑(alternativeBlocks at: self) value]
            ifFalse: [↑aBlock value]
```

Table 1: *Template showing additions to existing class* Number.

directly; we would get simple arithmetic loops by using do: with an Interval, a kind of collection that represents a bounded arithmetic progression. Using do: is convenient when we know that we want to look at all the elements of the collection, and do the same thing to each one. For example IndexedCollection might implement do: as shown in table 3.

However, if we want to retain more flexibility in controlling the generation process, there is a better way. We define the notion of a *supplier*, which will deliver the elements of a collection one at a time in response to messages. The protocol (set of messages and their intended meanings) for suppliers consists of the messages:

s atEnd

class name (existing)	BlockContext
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	
constructing **, aBlock** | | ↑BlockCollection with: self with: aBlock *accessing* **size** | | "Behave like a BlockCollection with self as the only element" ↑1 **at: index** | | "Behave like a BlockCollection with self as the only element" index = 1 ifTrue: [↑self]. self error: 'Subscript out of bounds'	
class name (existing)	BlockCollection
superclass	IndexedCollection
instance variable names	"none defined here"
class messages and methods	
"none defined here"	
instance messages and methods	
constructing **, aBlock** | | ↑self add: aBlock	

Table 2: *Templates showing additions to existing class* BlockContext *(2a) and the creation of a class template for class* BlockCollection *(2b).*

which returns true if there are no more elements to be supplied, and:

```
s next
```

which returns the next element. Now we can build our do: operation as shown in table 4. Then each kind of collection needs to implement the message asSupplier, which returns an appropriate supplier. Table 5 shows what a supplier might look like for IndexedCollection (including its creation). If an attempt is made to read past the end, position will be incremented beyond the size of the collection, and next will provoke an error when the at: tries to

class name (existing)	IndexedCollection
superclass	Collection
instance variable names	"none added here"
class messages and methods	

"none added here"

instance messages and methods

```
enumeration
do: aBlock
    | index limit |
    index ← 1.
    limit ← self size.
    [index < = limit] whileTrue:
        [aBlock value: (self at: index).
        index ← index + 1]
```

Table 3: *Template showing additions to existing class* IndexedCollection.

class name (existing)	Collection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	

"none added here"

instance messages and methods

```
enumeration
do: aBlock
    | supplier |
    supplier ← self asSupplier.
    [supplier atEnd] whileFalse:
        [aBlock value: supplier next]
```

Table 4: *Template showing additions to existing class* Collection.

access an element beyond the size. An alternative approach, which gives a more useful error message at the expense of duplicating a check that at: must perform anyway, is to define next as follows:

```
next | |
    position > = collection size
      ifTrue:
        [self error: 'Attempt to read beyond last
          element'].
    position ← position + 1.
    ↑ collection at: position
```

Similar supplier classes would be needed to provide generation capability for all of the different kinds of Collections.

With the supplier approach to generators, we can easily build a loop that sequences through two collections in parallel (see table 6). This would be very difficult if we did not have suppliers, but made collections implement do: directly. The problem is that while we could use do: to get one of the two collections to deliver its elements to a block of our choosing, there would be no way to get the other collection to deliver exactly one element each time the block is invoked.

Suppliers are so useful as a concept and as a protocol that Smalltalk actually includes them, under the name of ReadStream. The important point is that we could have built them ourselves if the system implementors hadn't gotten there first.

Although the only kind of supplier we have constructed is one that sequences through a collection, other kinds of suppliers are possible: they just have to respond appropriately to atEnd and next. For example, one could imagine a supplier that selected elements at random from a collection in response to next.

Exceptional Conditions

One of the difficulties in designing programs that (at least appear to) work reliably is designing the control structures for handling "infrequent" events. An infre-

class name (existing)	IndexedCollection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	
enumeration **asSupplier** | | ↑ IndexedCollectionSupplier of: self	

class name	IndexedCollectionSupplier
superclass	Object
instance variable names	collection position
class messages and methods	
creation **of: aCollection** | | ↑self new of: aCollection	
instance messages and methods	
creation **of: aCollection** | | collection ← aCollection. position ← 0 *accessing* **atEnd** | | ↑position > = collection size **next** | | position ← position + 1. ↑collection at: position	

Table 5: *Templates showing additions to existing class* IndexedCollection *(5a) and the creation of a class template for class* IndexedCollectionSupplier *(5b).*

quent event is any event which is (a) qualitatively different from what happens most of the time and (b) not so common that one wants to test for it in the normal flow of control. One example of an infrequent event is an additional exit from a loop. Suppose we would like to write a searching loop that finds the maximum element of a collection of non-negative numbers but stops searching if it finds an element greater than 1000. Such a loop might be implemented as shown in table 7.

We want the block [[supplier atEnd] ...] to respond to the withExit message by giving back a blocklike object which we can assign to the variable theLoop. The ability to name this object allows us to exit from it midcourse. These BlockWithExit objects (see table 8) need to remember only two pieces of information: the original block, to execute in response to the value message, and where to send control if an exit message is sent.

The original statement theLoop ← ... doesn't actually

class name (existing)	Collection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
enumerating
with: anotherCollection do: aBlock
     | mySupplier itsSupplier |
   mySupplier ← self asSupplier.
   itsSupplier ← anotherCollection asSupplier.
   [mySupplier atEnd or: [itsSupplier atEnd]]
      whileFalse:
         [aBlock
            value: mySupplier next
            value: itsSupplier next]
```

Table 6: *Template showing additions to existing class* Collection.

class name (existing)	OrderedCollection
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
searching
maxBefore1000
   | supplier max value theLoop |
   max ← 0.
   supplier ← self asSupplier.
   theLoop ←
      [[supplier atEnd]
         whileFalse:
            [value ← supplier next.
            value > 1000 ifTrue: [theLoop exit].
            max ← max max: value]] withExit.
   theLoop value. "Actually do the loop block"
   ↑max
```

Table 7: *Template showing additions to existing class* OrderedCollection.

class name	BlockWithExit
superclass	Object
instance variable names	block exitBlock
class messages and methods	

```
creation
with: aBlock | |
   ↑ self new with: aBlock
```

instance messages and methods

```
creation
with: aBlock | |
   block ← aBlock

control
value | |
   exitBlock ← [↑ nil]. "Exit to my caller if the block ever
   sends me the exit message"
   ↑ block value "Actually do the computation"
exit | |
   exitBlock value "Exit from the computation to the caller
   who sent the value message to me in the method just above"
```

class name (existing)	BlockContext
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
control
withExit | |
   ↑ BlockWithExit with: self
```

Table 8: *Templates showing the creation of a class template for class* BlockWithExit *(8a) and additions to existing class* BlockContext *(8b).*

execute the loop: it creates a block whose code is [supplier atEnd] ... . This block becomes the block variable of a new BlockWithExit as a result of the withExit message being sent. theLoop is set to the BlockWithExit just created. When theLoop is sent the message value, the value method in BlockWithExit first creates another block, the exitBlock, which, if evaluated, will return to the sender of value *regardless of how many other activations have intervened*. The value method in BlockWithExit then sends value to the original block, causing it to execute. If no exit is sent, the loop completes normally. If an exit is sent, the exitBlock is evaluated and control returns to the last statement of maxBefore1000, just as if the loop had completed.

Dynamic Binding

Another common kind of infrequent event is a request for information. For example, suppose we want to specify a default directory for disk files throughout some part of a program. We could pass this information as an argument through all intervening calls, but this would place an added burden (in time, space, and complexity) on many parts of the program that have no interest in this information. An alternative would be to set a global variable before starting the computation, and reset it afterwards; unfortunately, if the computation is interrupted (say by something like the loop exit construct we described earlier), this leaves the variable with the wrong value. Ideally, we would like to set up a structure that will get control if the default information is ever needed, without getting in the way of the rest of the program. Such an arrangement is called *dynamic binding*. We will illustrate how it can be used both for data and control.

Suppose we want to write something such as the following:

```
#defaultDirectory bindTo: 'Smith' in:
   [someComputation]
```

and then have the file system be able to ask for the current default directory by:

```
#defaultDirectory binding
```

Since we want the binding of defaultDirectory to 'Smith' to last only for the duration of someComputation, it follows that in order to find the binding of a dynamic variable, we must examine the data structures that Smalltalk uses to represent the state of a computation. In

class name	Binding
superclass	Association "Provides key and value variables, and messages for accessing them"
instance variable names	"none defined here"
class messages and methods	
creation **of: aSymbol to: aValue in: aBlock** | | ↑ self new of: aSymbol to: aValue in: aBlock	
instance messages and methods	
initialization **of: aSymbol to: aValue in: aBlock** | | key ← aSymbol. value ← aValue. ↑ aBlock value "Actually does the computation"	
class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	
binding **bindTo: value in: aBlock** | | ↑ Binding of: self to: value in: aBlock	

Table 9: *Templates showing creation of a class template for class* Binding *(9a) and additions to existing class* Symbol *(9b).*

class name (existing)	Symbol
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	
"none added here"	
instance messages and methods	

```
binding
binding | context |
  context ← thisContext. "Start here. thisContext is a
  machine register"
  [context = nil] whileFalse:
    [((context receiver isMemberOf: Binding)
        and: [context selector = #of:to:in: "Is it a
        binding..."
      and: [context receiver key = self]])  "...of this
      variable?"
        ifTrue: "Yes, return its value"
         [↑ context receiver value]
        ifFalse: "No, go on to the next context in the
       chain"
         [context ← context sender]].
  self error: ('No binding for' concatenate: self )
```

Table 10: *Template showing additions to existing class* Symbol.

particular, even though many messages may be sent in someComputation before the file system needs to find the binding of defaultDirectory, there must be some way to search the stack of methods that have been started but not completed, looking for whatever represents the binding of defaultDirectory. In Smalltalk, each element of this stack is a MethodContext object, and the variable in a MethodContext that refers to its caller is called its *sender*. So searching this stack just means checking the current context's sender, its sender, and so on, until we find a binding of the variable. We know we have found a binding when we recognize a MethodContext in which the receiver of the message is a Binding (see tables 9a and 9b), and which was created in response to a particular message. During this computation (↑ aBlock value in table 9a), a MethodContext will exist in which the receiver is the Binding and the message is of:to:in:. This is how we recognize a binding in the stack of MethodContexts. The searching process is shown in table 10.

Note that by combining dynamic binding with the ability to name exit points (eg: by doing #theExit bindTo: to create a BlockWithExit), we can arrange for dynamically bound exceptional events to stop a computation in midstream. More complicated arrangements that allow the parts of the computation being stopped to clean up after themselves are also easy to construct.

Coroutines

Generator loops are an example of *producer/consumer*

structures: the supplier produces elements, and the program that invoked the loop construct consumes them. As we saw earlier, one way to do this is to assign responsibility as follows:

```
Producer
implements:           do: aBlock
delivers values by:   aBlock value: theNextElement

Consumer
receives values using: [:elementName|
                          dosomethingToTheElement]
```

Under this arrangement, the producer can use any desired control structure internally, just by sending the message:

```
aBlock value: theNextElement
```

to the block whenever a new element has been generated; the consumer, however, is confined to executing the same block for each element. The other arrangement reverses the situation:

```
Producer
implements:           atEnd, next
delivers values by:   returning a value from next

Consumer
receives values using: producer next
```

Under this arrangement, the producer has to use instance variables, rather than control variables, to remember what state it is in, but the consumer can call for new elements using any control pattern it wants.

The control structure *coroutines* allows *both* the producer and consumer to use any control pattern. Notice that in the first arrangement, the producer has to retain its argument aBlock to be able to send it value: for each element; in the second arrangement, the consumer has to retain the producer to be able to send it next for each element. In the coroutine arrangement, both sides retain a common object called a *port*. The purpose of the port is to remember the control state of one partner while the other partner is running. Let us now build a port in which the consumer invokes the producer with the messages next and atEnd, and the producer invokes the consumer with the messages nextPut: anElement and markEnd. A loop in this implementation might look similar to the following:

```
Consumer
 | first second |
 portForProducer ← someCollection asProducer.
 "Here is a sample loop that takes elements two at a time"
 [portForProducer atEnd]
     whileFalse:
         [first ← portForProducer next.
         second ← portForProducer next.
         "Do something with first and second"]

Producer Collection
asProducer | port |
     port ← Port new.
     port producer: [CollectionProducer of:
        self with: port].
     "Create a new process for the producer"
     ↑ port

CollectionProducer
of: aCollection with: portForConsumer | |
     "Here is a sample loop that generates elements three at a
     time"
     [someCondition]
        whileTrue:
           [portForConsumer nextPut:
              someComputation1.
           portForConsumer nextPut:
              someComputation2.
           portForConsumer nextPut:
              someComputation3].
     portForConsumer markEnd
```

The code in both consumer and producer can involve any combination of loops, messages, or other control structures: the consumer can request a new element at any time with portForProducer next, and the producer can deliver an element any time it has control with port-

ForConsumer nextPut: anElement. Interleaving a consumer that wants pairs of elements with a producer that generates triplets is a very simple example of the freedom that both partners enjoy in this arrangement.

To implement Port we need to consider how Smalltalk allows us to get hold of our current control state, since whenever control goes from consumer to producer or vice versa, we have to save the state of the partner that is giving up control. For just such purposes, Smalltalk provides a primitive notion of a *process*, an entity which has its own control state and can be suspended and resumed. The usual way to create a new process is with:

```
aProcess ← [someComputation] newProcess.
```

The process can then be started up by:

```
aProcess resume
```

and it will compute "in parallel" with the current computation until it finishes someComputation or it (or some other process) executes:

```
aProcess terminate
```

which stops it midflight. Alternatively:

```
[someComputation] fork
```

creates and starts an unnamed process that will proceed until the computation finishes.

class name	Port
superclass	"none added here"
instance variable names	consumerSemaphore producerSemaphore nextElement endMark
class messages and methods	
"none defined here"	
instance messages and methods	

```
initialize
producer: aBlock | |
    "Assume we are running in the consumer process, so create a
    new process for the producer."
    endMark ← false.
    consumerSemaphore ← Semaphore new.
    producerSemaphore ← Semaphore new.
    producerSemaphore signal. "So producer will proceed
    the first time"
    aBlock fork
consumer
next | anElement |
    consumerSemaphore wait. "Wait for producer to deliver
    an element"
    endMark ifTrue: "No more elements"
        [self error: 'Attempt to read past last element'].
    anElement ← nextElement.
    producerSemaphore signal. "Restart producer"
    ↑ anElement
atEnd | |
    consumerSemaphore wait. "Wait for an element or end
    mark"
    consumerSemaphore signal. "Doesn't consume the ele-
    ment"
    ↑ endMark

producer
nextPut: anElement | |
    producerSemaphore wait. "Wait for consumer to have
    taken last element"
    nextElement ← anElement.
    consumerSemaphore signal "Restart consumer"
markEnd | |
    producerSemaphore wait.
    endMark ← true.
    consumerSemaphore signal
```

Table 11: *Class template for class* Port.

To allow processes to synchronize their control or their use of data in an orderly way, Smalltalk provides *semaphores*. A semaphore logically represents the current availability of a finite resource: aSemaphore signal indicates that one unit of the resource has just become available, and aSemaphore wait indicates that the currently running process needs to take one unit of the resource and must wait if none is available (presumably until some other process does aSemaphore signal). A useful special case of this is a semaphore that always holds either 1 (meaning a resource is available) or 0 (meaning it is unavailable).

As an aside, we note that semaphores could have been implemented in Smalltalk (ie: not as primitive entities) at a considerable cost in performance: we only need the ability to temporarily guarantee that no other process could run aside from the one currently running (on this processor in a multiprocessor system). Smalltalk provides semaphores at a primitive level because they are such a help in building multiprocess systems that we wanted people to feel free to use them without worrying about their cost.

Given processes and semaphores, we are ready to implement Port (see table 11). The producer and consumer will each run in a process of their own, and we will use semaphores to make sure only one of them is running at a time. (The reader can easily imagine and might enjoy thinking about a version of coroutines which allows the producer to "get ahead" of the consumer. This requires a queue between the two, like the SharedQueue we will develop later.) The "resource" controlled by the semaphores will be free access to the variables in the port, nextElement and endMark, under the following arrange-

Consumer	Producer
"In portForProducer next:" consumerSemaphore wait. "Semaphore started with 0, consumer waits."	
	"In port nextPut:" producerSemaphore wait. "Semaphore started with 1, now has 0" nextElement ← anElement. consumerSemaphore signal. "Semaphore started with 0, now has 1"
anElement ← nextElement. producerSemaphore signal. "Semaphore had 0, now has 1" ↑anElement	
	"nextPut: returns, producer proceeds." "Later, producer does another port nextPut:" producerSemaphore wait. "Semaphore goes from 1 to 0 again"

Table 12: *Dialog between consumer and producer objects using the* Port *defined in table 11.*

ment: when consumerSemaphore has a 1, it means nextElement has something in it (or endMark has been set) and the consumer needs to run; when producerSemaphore has a 1, it means nextElement is vacant and the producer needs to run. Notice that the next and nextPut: methods are very similar.

A partial trace through an exchange of control would look like the dialog shown in table 12. Note that if the producer reached the second wait before the consumer took the first element, the producer would wait until the consumer did the producerSemaphore signal. A full discussion of how semaphores should be used to produce minimum waiting, minimum process switching, and correct synchronization is beyond the scope of this article; one important and useful special case will be presented in the following section.

Monitors—Asynchronous Structures

Even in personal computer systems there are often reasons to allow for the possibility of several things happening "at once" (ie: not synchronized with each other beforehand). The best examples involve communication with other users. For example, your machine could be listening for incoming messages through a network connection. But even on an isolated personal machine, you would like to be able to start the system on a time-consuming project (like printing on a hardcopy device) and continue to do interactive work. As we saw before, Smalltalk provides the ability to create independent *processes* and set them going "in parallel," and provides

class name	Queue
superclass	Object
instance variable names	array writer reader
class messages and methods	

```
creation
new: size | |
    ↑ self allocate init: size
```

instance messages and methods	

```
    initialization
    init: size | |
        array ← Array new: size.
        reader ← 0.
        writer ← 0
    access
    removeFirst | |
        reader ← reader + 1.
        ↑ array at: reader
    addLast: anElement | |
        writer ← writer + 1.
        array at: writer put: anElement
```

Table 13: *Class template for an initial implementation of class* Queue.

Process A	Process B
reader ← reader + 1.	
	reader ← reader + 1.
↑ array at: reader	
	↑ array at: reader

Table 14: *Execution of the* removeFirst *method using the implementation of table 13.*

class name	SharedQueue
superclass	Object
instance variable names	array writer reader accessSemaphore
class messages and methods	

```
creation
new: size | |
    ↑ self allocate init: size
```

instance messages and methods	

```
initialization
init: size | |
    array ← Array new: size.
    reader ← 0.
    writer ← 0.
    accessSemaphore ← Semaphore new.
    accessSemaphore signal "Give it the baton"
```

Table 15: *Class template for class* SharedQueue.

semaphores for synchronizing their behavior.

From semaphores we can easily build a more useful construct, called a *monitor*. The purpose of a monitor is to allow several processes to communicate with a data structure without getting in each other's way; failing to provide for this is another common source of bugs—consider the simple-minded implementation of a queue given in table 13. (The reader should ignore the obvious bugs: there is no check for an empty queue or for exceeding the size of the array.)

Suppose two processes both try to remove an element at about the same time, and the removeFirst method gets executed as shown in table 14 (the flow of time is vertical down the page, interleaving the statements executed by process A in the left column and process B in the right). One element is skipped—and one is returned twice! The solution to this problem is to consider "permission to update the state of the queue" as a resource that only one process can hold at any given time, like the baton in a relay race. So we can construct a safe Queue by giving it a semaphore that starts out with one unit of the resource (see table 15).

A pattern we will encounter in the implementation of SharedQueue will be to reserve a resource during the execution of a piece of code:

```
someSemaphore wait.       "Acquire the resource"
someComputation.
someSemaphore signal.     "Release the resource"
```

The code someComputation is called a *critical section*. We would like to be able to write the previous code fragment as:

```
someSemaphore critical: [someComputation].
```

class name (existing)	Semaphore
superclass	"none added here"
instance variable names	"none added here"
class messages and methods	

```
"none added here"
```

instance messages and methods	

```
critical sections
critical: aBlock | result |
    self wait. "Acquire the resource"
    result ← aBlock value. "Do the computation, save the
result"
    self signal. "Release the resource"
    ↑ result "Return the result of the computation"
```

Table 16: *Template showing additions to existing class* Semaphore.

The system actually provides this message to Semaphore, with a straightforward implementation which is shown in table 16. It is then easy to appropriately modify the two messages in SharedQueue (see table 17).

If two processes try to access the queue, the interchange shown in table 18 occurs (with a few steps left out). Note that the variable anElement is a local variable, and since the two processes have different contexts (despite the fact that they share the same instance of SharedQueue in this example), the variable anElement in Process A is different from anElement in Process B.

class name	SharedQueue
superclass	"none defined here"
instance variable names	"none defined here"
class messages and methods	
"none defined here"	
instance messages and methods	

```
access
removeFirst | anElement |
    ↑ accessSemaphore critical: "Reserve access for the duration of the block"
        [reader ← reader + 1.
        array at: reader]
addLast: anElement | |
    accessSemaphore critical: "Reserve access for the duration of the block"
        [writer ← writer + 1.
        array at: writer put: anElement]
```

Table 17: *Class template for class* SharedQueue.

Final Comments

Many languages don't have the flexibility we've just described; others, such as assembly language, have great flexibility at the expense of readability. What is it about the Smalltalk-80 language and system that makes all of the foregoing both possible and fairly readable? Three things come to mind:

- ●The existence of blocks, with and without arguments, and the simple square-bracket notation for writing them. This makes it possible to pass a piece of code to the implementor of a control structure, which can then execute the code whenever and however it is appropriate. ALGOL and LISP have constructs which capture some, but not all, of the power of blocks.
- ●The ability to manipulate the control state directly, as in the dynamic binding example. Of course disaster can result if you aren't careful, but a challenge like this is necessary to exploit the full power of your imagination. InterLISP (a widely used LISP dialect) has facilities which capture some of the power of Smalltalk in this area.
- ●The accessibility of the entire system to modification. Several of the examples we've described involve adding messages to fundamental classes like Object and BlockContext. Restraint is important here too. Several LISP systems derive tremendous power from this kind of openness.

Of course, we pay a price for all this flexibility and simplicity. A discussion of the time and space cost of blocks, visible control state, and a completely accessible system is beyond the scope of this article; we will just observe that the elementary instructions which implement control structures (branch, call, and return) take about the same proportion of the total execution time in a typical Smalltalk-80 implementation as they do in more conventional languages that don't use globally optimizing compilers.■

Process A	Process B
accessSemaphore wait "Semaphore now has 0 units of resource" reader ← reader + 1.	
	accessSemaphore wait. "Waits here"
anElement ← array at: reader. accessSemaphore signal. "Process B can proceed now, but immediately reacquires the semaphore"	

	reader ← reader + 1.
	anElement ← array at: reader.
	accessSemaphore signal.
	↑ anElement

↑ anElement	

Table 18: *Execution of the* removeFirst *method using the implementation of table 17.*

Is the Smalltalk-80 System for Children?

Adele Goldberg and Joan Ross
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

For many years our work on the Smalltalk project has carried with it the purpose of creating new technologies that can be used effectively for instruction, both to teach programming and to support the implementation of educational activities. While the Smalltalk-80 system is not specifically designed for school-age children, most of the applications that we developed as tests of the earlier Smalltalk systems were.

This article will present a brief history of the development of the Smalltalk-80 system that focuses on the instructional uses of its various predecessors. A significant part of this history is the redesign of the language syntax. Programming in Smalltalk involves creating a language for communicating among objects; this language is created within the syntactic restrictions of the Smalltalk-80 system. Often the programmer adds an additional level of syntax in which the language for communicating among objects is presented in terms of graphic images. An example of an instructional activity, the *Dance Kit*, illustrates the idea of such a language. Its design was motivated by the rich support for generalization and interactive graphics available in the Smalltalk-80 system.

Our original intention in writing this article was to *disabuse* readers of the idea that the Smalltalk-80 system, like LOGO, is a language for children. We concluded, however, that the other articles in this issue and the two books on the system (see references) will easily accomplish that task. It remains, then, for us to comment on the style of use of the system that our instructional work has taught us. Although there are a few places where knowledge of the Smalltalk-80 system is helpful, this article does not, in general, require such knowledge.

Learning to Program in Smalltalk

Initially when we ventured out into the schools to teach programming classes, we used a version of Smalltalk known as Smalltalk-72 (see reference 3). Our purpose in teaching these classes was threefold. First, we wanted to know if the language was teachable. In particular, we wanted to devise an appropriate pedagogical approach that could provide feedback on the design of the user interface as well as a basis for language redesign. And we wanted to begin to find out if software based on the concepts of objects and message-passing offered something special in the way of problem-solving tools for children and adults alike. The outcome of these investigations reinforced the value of the semantics of Smalltalk: that is, from the point of view of supporting computer-based problem solving, we found that the ability to organize information into objects that can be independently explored and linked together to create new kinds of behavior is a powerful computational tool.

Smalltalk-72 took the approach that the syntax was defined by the receiver of the message: the receiver read as much of the message as the receiver's method determined and then passed control to the next remaining token, which was seen as the receiver of the remainder of the message. This design came out of our assumption that the system user should have total flexibility in making the system be, or appear to be, *anything* that the user might choose. However, this meant that the only way that a reader could understand an expression was to execute the methods in his head. Furthermore, if a human could not parse an expression without executing the methods, the system itself would not be able to parse it. Thus Smalltalk-72 was a purely interpretive system, and its performance suffered accordingly.

The syntax design (or lack of it) was an example of taking the "flexibility" position to an extreme. Our experience in teaching Smalltalk-72 convinced us that overly flexible syntax was not only unnecessary, but a problem. In general, communication in classroom interaction breaks down when the students type expressions not easily readable by other students or teachers. By this we mean that if the participants in a classroom cannot read each other's code, then they cannot easily talk about it. Our intention was that the Smalltalk system serve as a communication mediator, but the lack of communication due to the runtime parsing of expressions was hindering this goal.

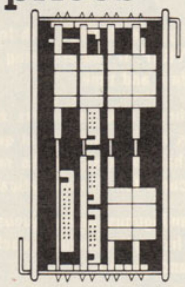
The Smalltalk-76 system took a stricter approach to syntax, making an incremental compiler a vital part of the system design. Expressions could be parsed by the human reader, although full understanding of the expressions required that the programmers choose identifiers and message names wisely. In this way, the programs could be read by other students or teachers. This approach to syntax remains in the Smalltalk-80 system.

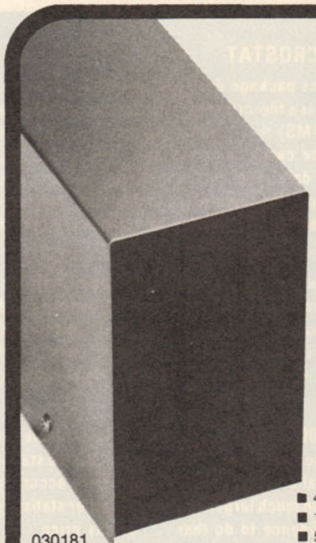
In Smalltalk, languages are designed whenever the programmer specifies the message protocol of a class description. These are the languages with which objects in the system share information among themselves as well as with the human user. Users can profit enormously from defining their own language, learning about their native language in the process of constructing another. In addition, the concept of classes and instances provides a uniform way to organize information in Smalltalk. Communication and organization of information are fundamental aspects of problem-solving activities. In this regard, the needs of school children match those of system designers.

In order to teach Smalltalk programming, the pedagogical approach we developed is to present a fully implemented model of something that the student can use and then modify. The idea is to help early learners grow accustomed to computer interaction and to the notion of sending messages in order to invoke behavior from existing objects. The students can then create two or more instances of an existing class and, through experimentation with the messages to which the class of objects responds, discover the similarities and differences among the instances. In this way, the students apply observation and hypothesis-generating skills while enjoying a highly interactive, graphical discourse with the system. This latter characteristic depends, of course, on the user's ability to provide enticing visual displays of the instances.

We can use this pedagogical approach for learners with varying levels of skills by adjusting the complexity of the initial model. Instruction proceeds by having the students learn to "read" the description of the model (that is, the code). They then modify it so that each existing instance demonstrates a new, shared behavior (ie: the student adds a message/method to the class description). At this level, we are teaching students fairly standard programming skills that involve sequencing of messages to objects. The concept of naming variables was previously explored as part of the process of creating instances of classes. It is further explored in declaring and using temporary variables in support of a method. Self-reference in the form of messages to the object denoted by self comes naturally and is not dealt with as extraordinary. The curriculum framework we follow involves:

- use of an already existing model
- reproduction of the model with some addition
- substitution into the model to produce a new result
- introduction of the model into new contexts (ie: using it as a component of some other example)

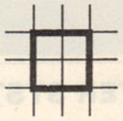




One introductory example we employed with elementary and junior high school students was a series of projects to use, modify, and extend the definition of a Box description. A box is an object that looks like a square: it can be drawn on the display screen or erased. It can grow bigger or smaller, and turn right or left, and it can be moved to different screen locations.

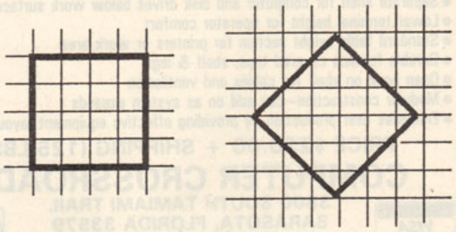
Suppose we create jill as an instance of a Box:

jill ← Box new.



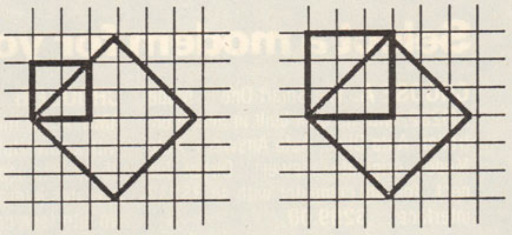
Then jill can grow and turn:

jill grow: 50.   jill turn: 45.

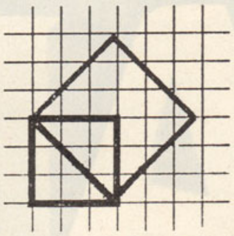


Many boxes can be created:

jan ← Box new.   jan grow: 25.



jan turn: 90.



Animations and pretty designs come from sending a sequence of grow:, turn:, and move: messages to the various boxes.

Once the students used several instances of Box, they modified the definition of Box in order to have all its instances follow the display screen cursor as the cursor was moved about by a pointing device. Generalizations of Box led to descriptions of triangles, hexagons, and other polygons. Simple games of "leap frog" or space war

were new contexts in which to place the geometric objects. Students discovered that judicious placement of geometric shapes formed pictures, and so "painting editors" were popular project areas for the students.

The resource-center approach we took in locating our computer system in a school emphasized shared projects, so that each student might bring a different skill to a project. Because we feel *design* is at least as important as implementation (programming), we encouraged students with good ideas for projects to act as resources for those students who preferred to write programs. Miniature research teams seemed to form in a natural way. The nonprogrammers on the team did their designs both visually (by sketching drawings of desired outcomes) and verbally. In the latter case, they designed by determining the needed objects and then specifying the language with which these objects would interact. The students benefited from the Smalltalk approach to description even before completing, or perhaps without completing, a running program.

After our experiences in the schools, we felt that further studies of graphical user interfaces were needed in order to improve the visual feedback Smalltalk provides as its programming interface. The Smalltalk-76 system was created primarily as a basis for implementing and studying various user-interface concepts. It gave the users, mostly adult researchers, further ability in refining existing classes through the use of subclassing. This meant that the programmer could now modify a running model without creating a change to already existing examples of that model. *Programming-by-refinement*, then, became a key idea in our ability to motivate our users.

Contrary to the idea that a computer is exciting because the programmer can create something from seemingly nothing, our users were shown that a computer is exciting because it can be a vast storehouse of already existing ideas (models) that can be retrieved and modified for the user's personal needs. Programming could be viewed and enjoyed as an evolutionary rather than a revolutionary act. The frustration of long hours of writing linear streams of code and then hoping to see some aspect of that code execute was replaced by incremental development. Emphasis was placed on learning how to make effective use of existing system components (objects in the Smalltalk sense). Much of the teaching we did was to show users how to search for and read the descriptions of the many useful components we and others (and even new users) continued to add to the system.

Fundamentally, the Smalltalk approach to software has exciting potential for educational use. But why only "potential"? As the system development work has proceeded from our initial work in the schools using Smalltalk-72, greater emphasis has been placed on providing a powerful system that is of interest to computer professionals as well as children. The Smalltalk-80 system, in its approach to providing a programming interface, focuses more on software development for the professional. But the basic design of the system remains that of collections of objects. There is a clear layering of these objects in terms of system versus user-interface support. Our success in bringing this system back into the classroom depends upon our ability to create a set of useful components (class descriptions) that the user can manipulate, as well as modify and combine, in order to create new components. Among the components already developed toward this goal are text and text editors, graphical images and "sketching" and animation editors, as well as "browsers" for seeking out other, already existing components from libraries of such information. We have begun, but we still have a great deal of work ahead of us to design and store in libraries the viewing and controlling components of graphical user interfaces.

Kits for Instructional Activities

So far we have commented on the use of the Smalltalk-80 system for programming. In doing so, we have placed a great deal of importance on the existence of a library of components. Such a library is needed for both professionals and nonprofessionals. In order to improve the system for educational use, better support is needed to assist computer-based curriculum designers in developing flexible instructional activities (perhaps in the form of a special library). Several examples of instructional activities that have been implemented in Smalltalk-72 or Smalltalk-76 are described in Laura Gould and William Finzer's "A Study of TRIP: A Computer System for Animating Time-Rate-Distance Problems", (see reference 5) and Adele Goldberg's "Educational Uses of a Dynabook" (see reference 2).

More recently, we have been trying to work out the idea of a kit for constructing such activities. By a "kit" we mean a set of components and a set of tools (by means of which these components can be viewed and manipulated) that can be used to create many different but related things. Thus, the VisiCalc program (see reference 1) can be viewed as a kit for making business forms; any text editor is a kit for creating textual documents, and any "painting" system such as the Smalltalk ToolBox is a kit for making sketches (see "ToolBox: A Smalltalk Illustration System," by William Bowman and Bob Flegal, on page 369 of this issue).

For developing instructional activities, we believe that a kit can be used as an interface to hide the unnecessary details of the Smalltalk-80 system. A kit could provide a (possibly graphical) interface to the system for the user (student, teacher or curriculum developer) who prefers to focus attention on only one or two aspects of the system. In such a kit, we maintain the Smalltalk approach of selecting objects that respond by receiving a message. The experience gained using one level of the system can be applied to learning successively lower levels. Of utmost importance, the code that implements the kit should be accessible at the next level of interface so that the kit can act as a starting point for further refinement and instructional design.

For the most part, the kits we have designed create new programming interfaces. Most came about by looking at instructional activities from other systems and seeing which ones we liked. We then used the concepts of Smalltalk classes and instances in order to help us generalize the idea of the activity into kit form so that a teacher or student would be able to create personal variations of the activity. The remainder of this article presents an example of a kit that could be implemented in the Smalltalk-80 system.

Invitation to the Dance: Prelude

Imagine that you are a choreographer, able to direct the movements of a dancer on the stage. As the dancer follows your instructions and you see their effect, you may modify them, partly to more closely fulfill your initial images, and partly because observing the actual execution may give rise to new creative ideas.

Since you probably don't have access to a real stage and a real dancer, imagine that your computer's video screen is the stage on which you can direct the movement of a graphical dancer by means of a simple programming language. You can experiment with different sequences of instructions that direct your graphical dancer to replace parts of itself with other parts (thus raising and lowering its arms and legs) and to move in various directions across the screen. The system that allows you to create such dances is called the "Dance Kit."

Setting

The original idea for the Dance Kit came from Bill Finzer, a mathematics teacher at San Francisco State University. Bill was introducing a friend to a Commodore PET computer. Because both he and his friend are fans of an Indonesian dancer called Pak Jana, Bill conceived the idea of teaching his friend about programming in the context of choreography. He wrote a BASIC program called PAKJANA that allowed her to control the movements of a highly stylized dancer on the screen. (It has subsequently been used as a very successful introduction to programming for students in a course called "Computers without Fear.")

The PAKJANA figure is shown in figure 1. It is controlled by a sequence of commands, including a repetition construct. The commands either:

- replace a face, an arm, or a leg
- move the whole body up, down, to the right, or to the left
- create a pause (wait command)
- create a repetition of the commands

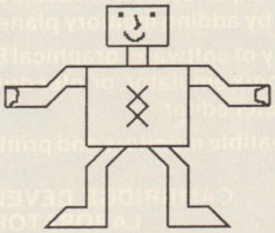


Figure 1: *The PAKJANA figure. Children were taught the basic ideas of programming by teaching this figure how to "dance."*

A large set of "replacement parts" is provided: there are nine different expressions and nine different positions in which any arm or leg may appear. The user specifies these replacement parts and the movements that the dancer can make by choosing from a list of commands in a very simple language. Thus, users can create programs that move the dancer in a predictable sequence of dance routines. (That users may initially find the effects of their programs not entirely predictable only adds to the fun.) The replacement parts for PAKJANA are shown in figure 2. (The program is available from Bill Finzer at the Center for Mathematical Literacy, Mathematics Department, San Francisco State University, 1600 Holloway, San Francisco CA 94132. The project was supported by an Academic Development Grant for the California State College System.)

Theme

Our Dance Kit evolved from Bill's BASIC program by considering possible extensions given the interactive graphics support of the Smalltalk-80 system. The goal of the Dance Kit is to provide a very flexible programming language by giving the user (the "programmer") the ability to draw the figure for which a dance can be choreographed. This figure not only moves about the screen, but also may change the position, size, shape, or color of its parts. One of the editing capabilities provided to the user of the kit is the ability to draw and subdivide the figure into parts. The user can then draw a set of images that replace each of the subdivisions. We call these replacement parts. They appear on the display screen as a part of the programming language the user can employ to create dance routines. An example figure is shown next to the label POSITIONS in figure 3. The user can view replacement parts of a particular subdivision by pointing on the screen to the part of the figure to be replaced. As an example, see the sequence of display screen views shown in figure 5 on pages 362 and 364.

The programming language also contains "steps" for placing the figure and "bridges" that allow repetition of some sequences of instructions. The steps, GO, TURN,

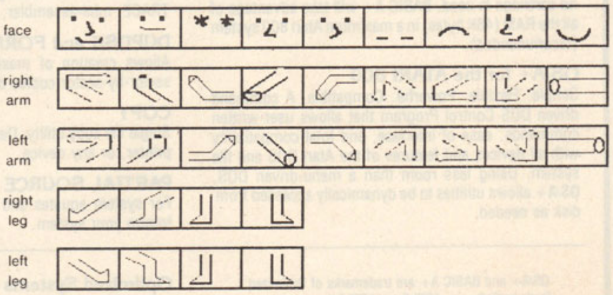
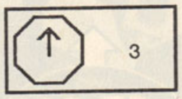


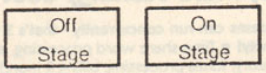
Figure 2: *Replacement parts for the PAKJANA figure. Combinations of these options allow the figure to be animated.*

and PAUSE, are given numerical parameters that indicate how large a step the figure should take. The user selects steps GO and TURN in cases where relative directional placement is desired. Alternatively, the user programs with a step that combines a direction and movement. For example, to take three steps in an upward (↑) direction, the step instruction is:

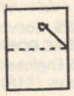


The user sets the arrow icon to specify the direction.

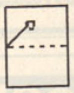
Bridge REPEAT is given a numerical parameter specifying the number of times a sequence of positions and steps should be repeated. Bridge REPEAT UNTIL is associated with a condition for terminating the repetition. We envision a fixed set of conditions such as:



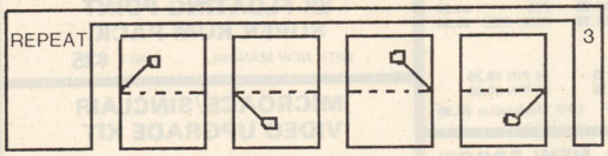
A GHOST bridge indicates that an image of the dancer should remain on the screen in the dancer's last position whenever the figure steps using the GO instruction. If replacement parts include:



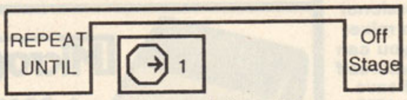
for the right arm, and:



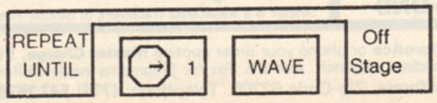
for the left arm, then a simple sequence to have the dancer wave each arm three times looks like:



Notice that the replacement parts can overlap. For example, the arm parts are large and overlap the head part so that it is possible to lift the arms above the head. Similarly, the leg parts allow overlap with arm parts. The bridge "covers" the steps to be requested, with a condition specified at the right girder. The figure can slowly dance stage right using:



The user is also able to define "dance routines" that enable certain fixed sequences to be named, stored, retrieved for further editing, and used as a "sub-routine." Suppose the first sequence of commands is stored as the Dance Routine known as WAVE. We can then use this routine in another one to get:



On the screen, one routine (the one being edited by the user) is active at a time. Controls for managing routines are:

- ●DELETE (delete the routine from the language)
- ●STORE (store the current dance act as this routine)
- ●NEW (start a new dance act)
- ●COPY (make a dance act exactly like the current one)
- ●EDIT (make the selected routine be the current one you are editing)

A new dance act has the initial name CURRENT DANCE, as shown in the figure 4.

The user creates a Dance Act by using these elements (steps, bridges, and routines) of the programming language "DanceTalk" as shown in figure 4. This is done by simply pointing at the desired element and moving it

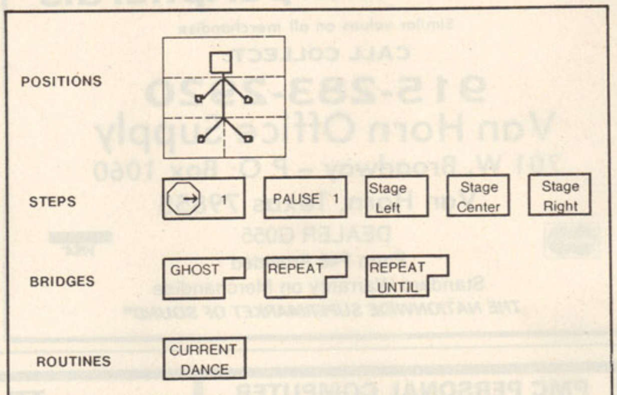


Figure 3: *A choreographer's programming language.*

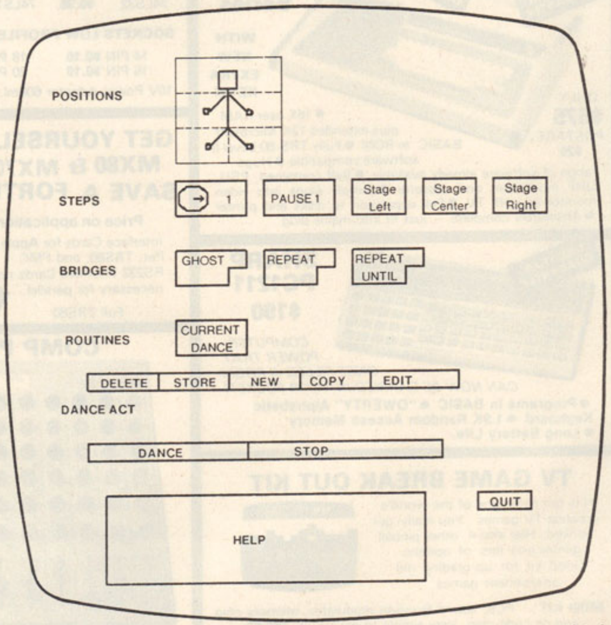


Figure 4: *A screen view of DanceTalk. This image, which appears on the video display of a Smalltalk system running the Dance Kit program, gives the user a menu of options with which to animate the "dancer."*

into the initially empty area in the lower part of the screen. Selected elements are highlighted by complementing their screen area. Notice the rectangular area labeled HELP at the bottom of the screen. When the user points into this area, a description of what to do next is shown, or a comment about a selected element is given.

First Variation: The Stick Person

A user of our Dance Kit might see the sequence of screen views shown in figures 5a through 5f. After a user has completed a longer Dance Act with two choreographed (sub-)routines KICK and JUMP, the screen might look like figure 5f. Now when the user indicates DANCE in the bottom menu of commands, the top part of the screen clears as if a curtain were rising, and the user sees the given sequence of views—an animation (see figure 6).

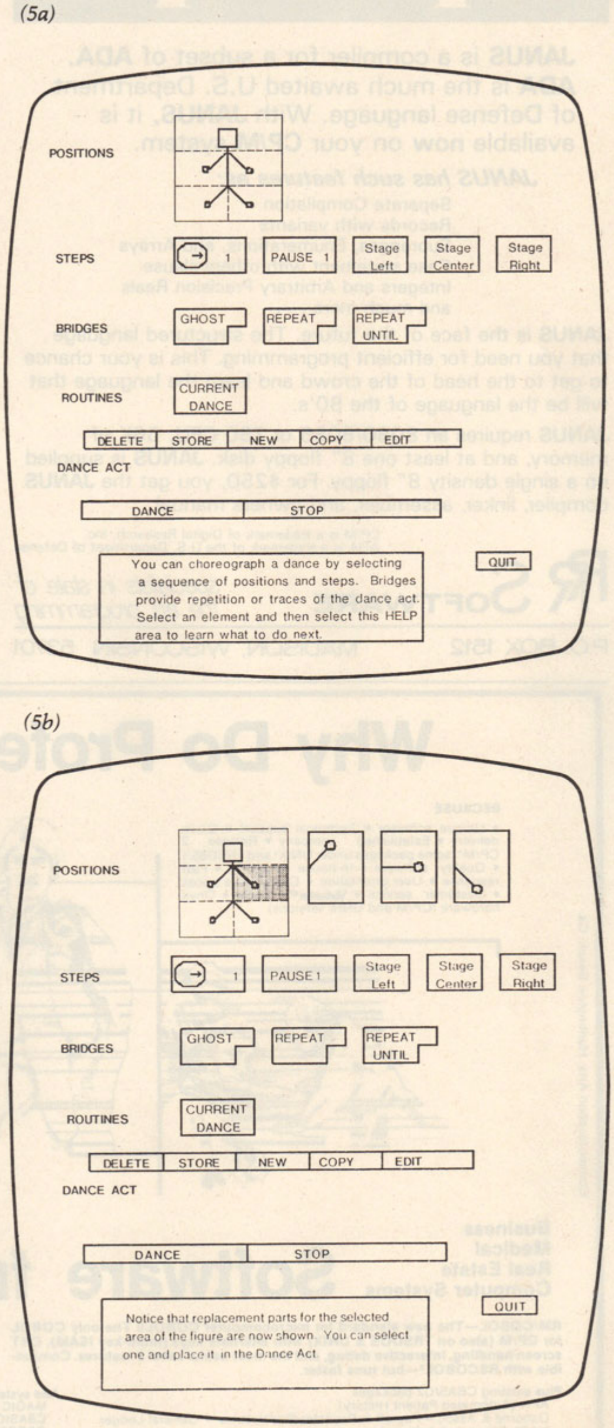
Note that a grid with a scale underlies both the space and time dimensions. These could be specified and experimented with by the user.

Second Variation: The Big Turtle

Our dancer may assume any size and shape we desire, and we can subdivide the dancer into any number of rectangular areas in order to create replaceable parts. The basic figure shown in figure 7 might be used. The dotted lines show the user's subdivision of the figure. Replacement parts for the section labeled D overlap section A, and might appear as shown in figure 8. Other replacement parts are left to the imagination of the reader.

Dance Acts can be shared by different figures, except that both replacement parts and routines will have to be

Text continued on page 365



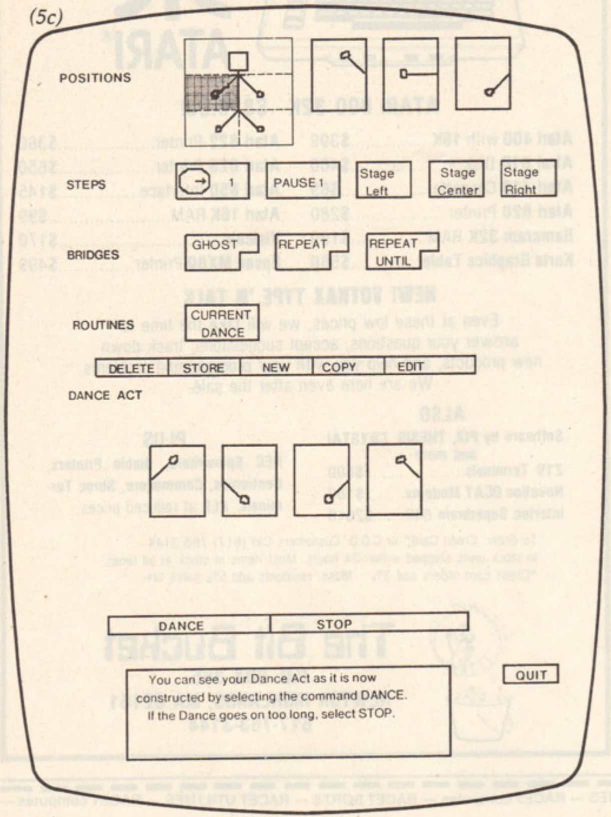
You can choreograph a dance by selecting a sequence of positions and steps. Bridges provide repetition or traces of the dance act. Select an element and then select this HELP area to learn what to do next.

Notice that replacement parts for the selected area of the figure are now shown. You can select one and place it in the Dance Act.

Figure 5: *Here and on page 364 are six views of the Dance Kit program during the creation of a dance. The "help" box, shown at the bottom of each figure, is always active. The shaded area indicates the item currently being worked on.*

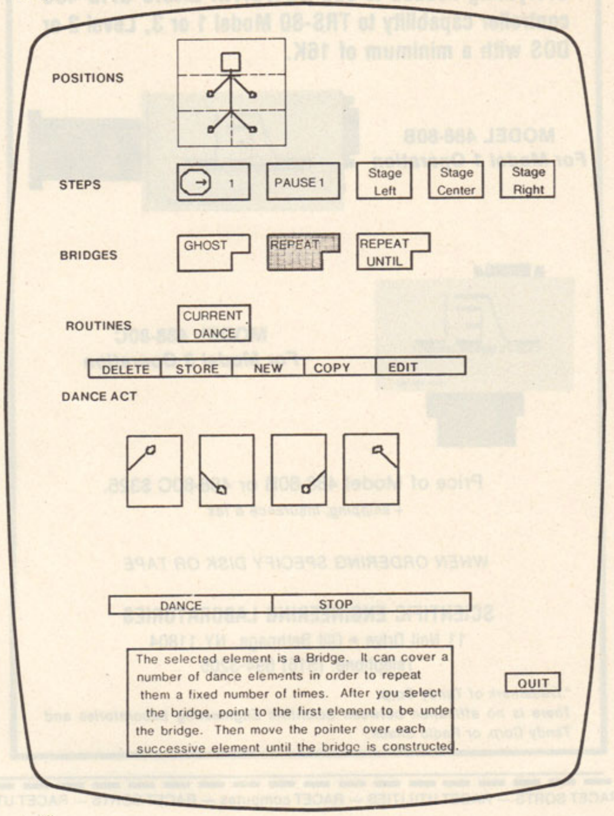
Figure 5 continued:

(5c)



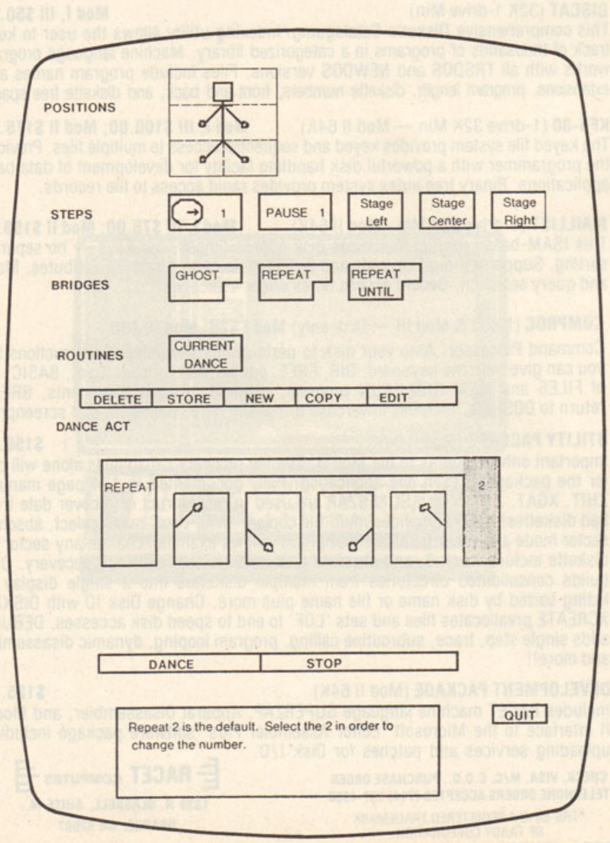
You can see your Dance Act as it is now constructed by selecting the command DANCE. If the Dance goes on too long, select STOP.

(5d)



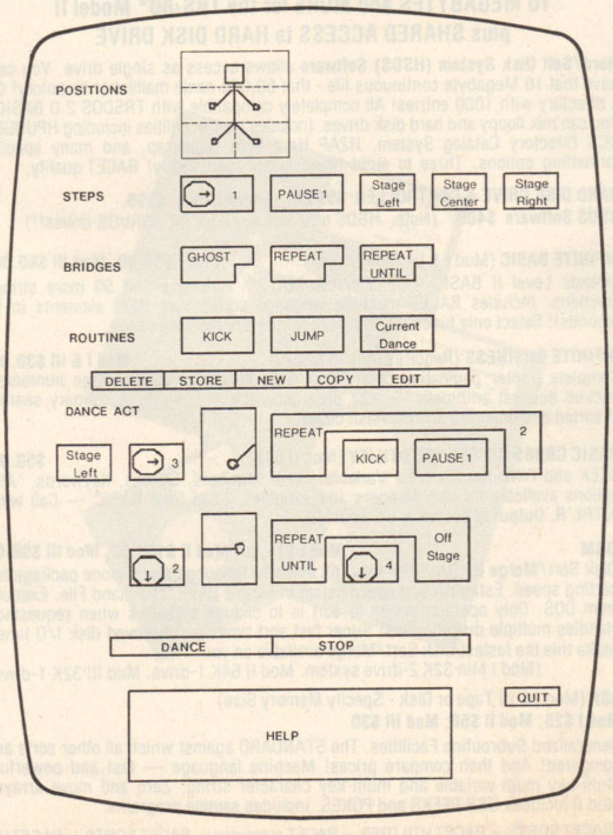
The selected element is a Bridge. It can cover a number of dance elements in order to repeat them a fixed number of times. After you select the bridge, point to the first element to be under the bridge. Then move the pointer over each successive element until the bridge is constructed.

(5e)



Repeat 2 is the default. Select the 2 in order to change the number.

(5f)



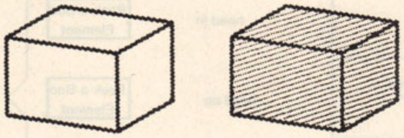
Text continued from page 362:

changed appropriately. A Turtle Dance Act that is akin to the Stick Person Dance Act shown in the first variation appears in figure 9. The animation for this Dance Act consists of the sequence of views shown in figure 10.

Third Variation: Boxes

The dancer might be a simple geometrical shape. The dancing needn't be subdivided, but replacement parts for the whole figure might be available.

For example, the user might create the following replacement parts:



Suppose the initial dance is set with the dancer moving toward the right. A possible Dance Act is shown in figure 11, where next HEX is defined as:

GO 1	TURN 60

The ghost is used to leave a trace of the box after each step. Each step unit was presumably scaled by the user to be the size of the box so that no overlapping occurs. At the end of this act the screen would look like figure 12.

Another possible geometric design comes from the building blocks shown in figure 13. If the dance act is defined as shown in figure 14, then at the end of the dance the screen will look like figure 15. When a figure of one color is superimposed on a figure of another color, the underlying figure disappears.

Fourth Variation: The Degenerate Turtle

The Dance Kit can be used to do conventional Turtle geometry (see reference 6) by allowing the figure to degenerate to a point (no replacement parts need apply) and defining a scale such that GO 1 means to go to the next point on the screen.

Dancing School

The Dance Kit is one example from a variety of kits and ideas for kits that we have entertained and that have entertained us. One of our major concerns is to create an environment in which the design of interesting and imaginative educational materials will be fostered, and we believe that the Smalltalk-80 system will make it easier to create such kits.

We have given much thought to some necessary characteristics of a framework for a Dance (or any other) Kit. We suggest that certain services always be present on the screen. For example, a help system is of supreme importance. We have provided an indication of the help system we would incorporate into the Dance Kit. The TRIP system for animating algebra word problems (see reference 5) provides such a complete HELP facility that even fearful users need be told only how to use the pointing device in order to control all the functionality of the system. Other possibilities for services include a LIBRARY, a GUIDE to other activities that might be appropriate, and a facility that allows users to enter sug-

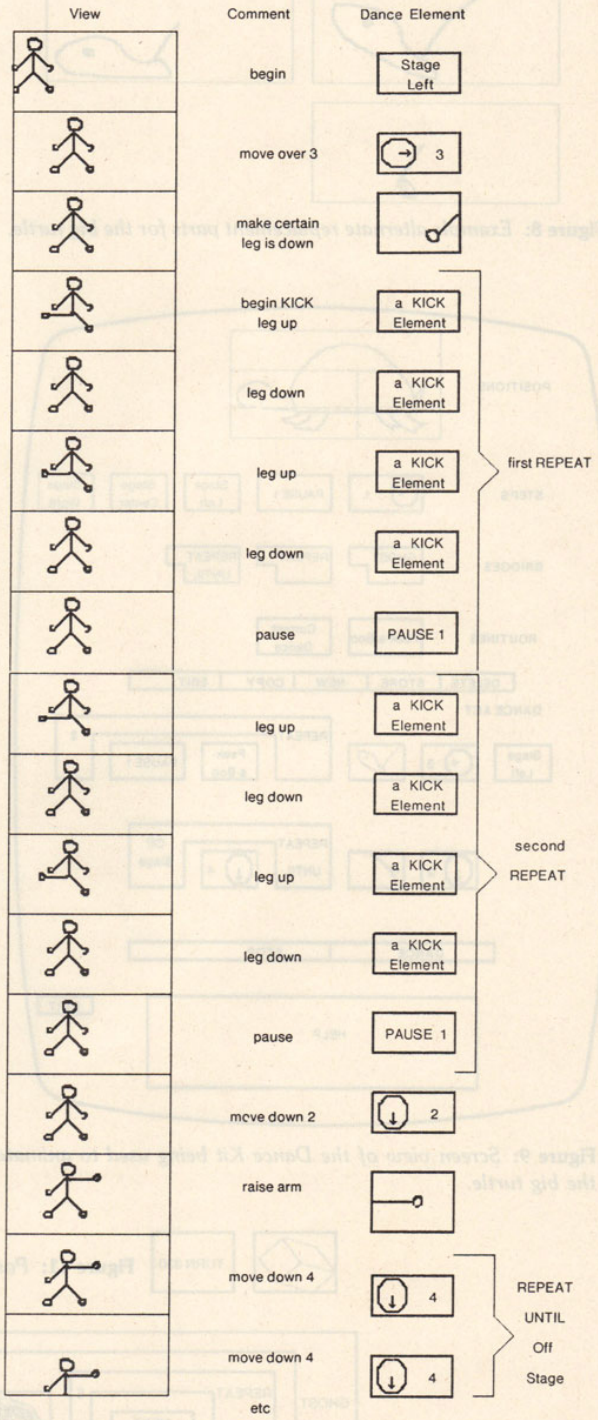


Figure 6: *Execution of the dance given in figure 5f.*

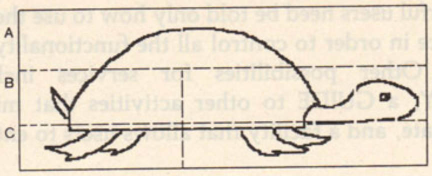


Figure 7: *A basic drawing of a big turtle that can be animated.*

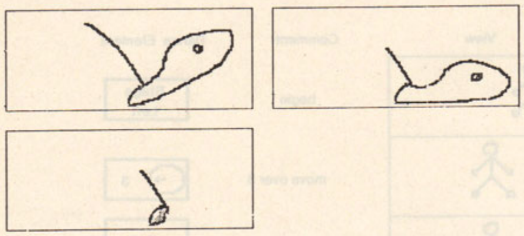


Figure 8: *Example alternate replacement parts for the big turtle.*

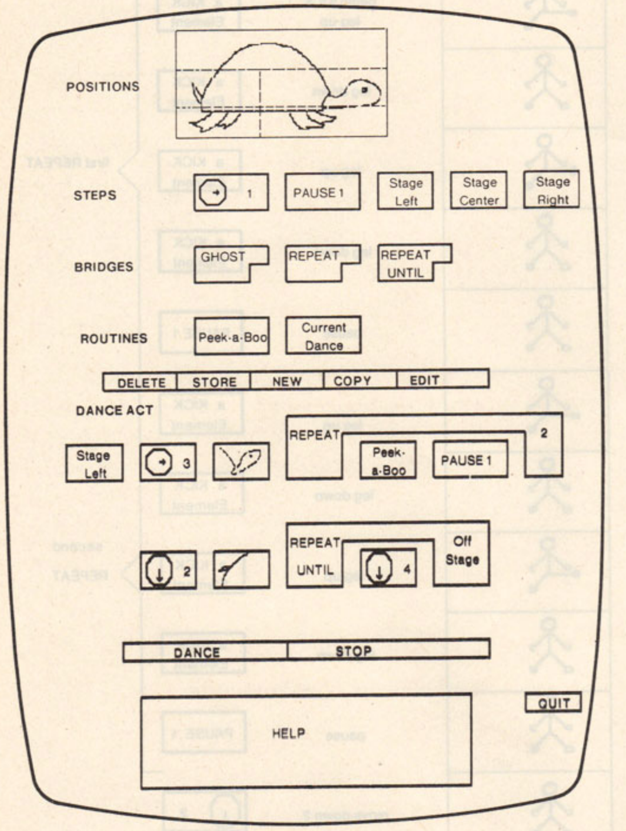


Figure 9: *Screen view of the Dance Kit being used to animate the big turtle.*

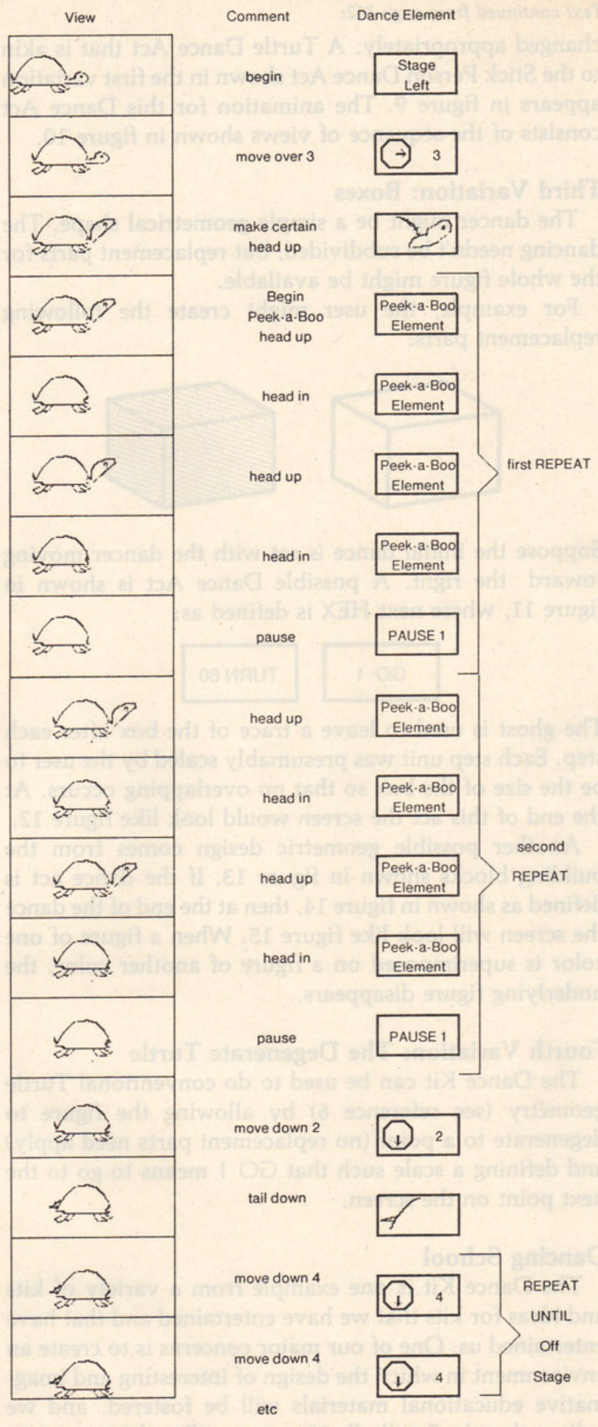


Figure 10: *Execution of the dance given in figure 9.*

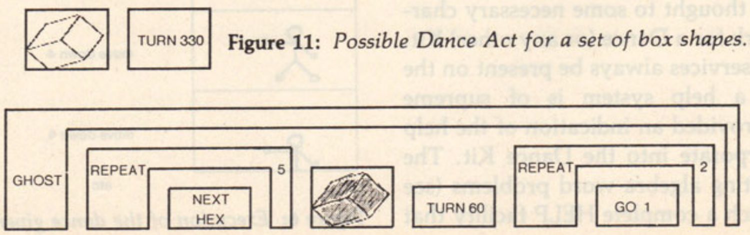


Figure 11: *Possible Dance Act for a set of box shapes.*

gestions and other feedback for the instructor (ie: a GRIPE).

We have also imagined a Dance Kit in a computerless classroom. The idea that computer-based activities should have concrete analogues, especially for young children, has been well received in the experience of the MIT LOGO group. We would use the children as the figures in the dances and create the Dance Act instructions and routines on paper or a blackboard or.... Of course if we want to leave ghosts, we will need to enlist the services of more than one child.

In this mode of use of the Dance Kit we are fond of imagining children as the design elements of the seven possible friezes shown in figure 16; each frieze is characterized by the group of transformations or symmetries that

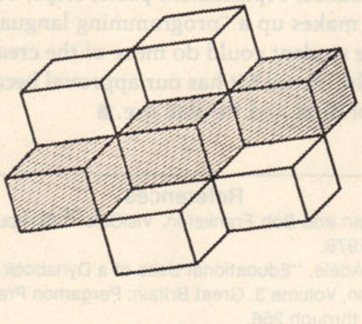


Figure 12: *Drawing made by the Dance Act of figure 11.*

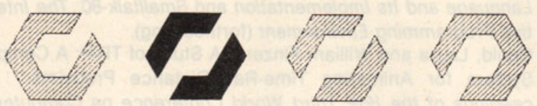


Figure 13: *A basic drawing of some geometric designs that can be used by the Dance Kit.*

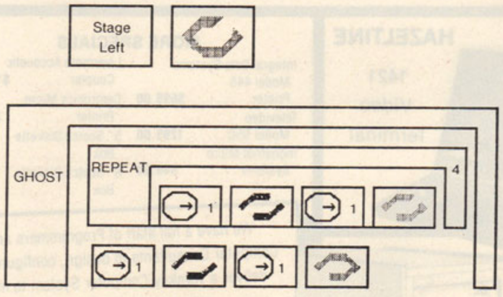


Figure 14: *Possible Dance Act for the set of geometric designs.*

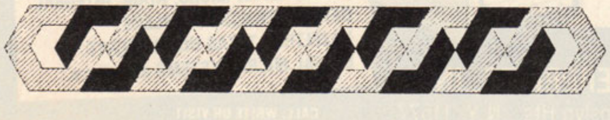


Figure 15: *A drawing made by the Dance Act of figure 14.*

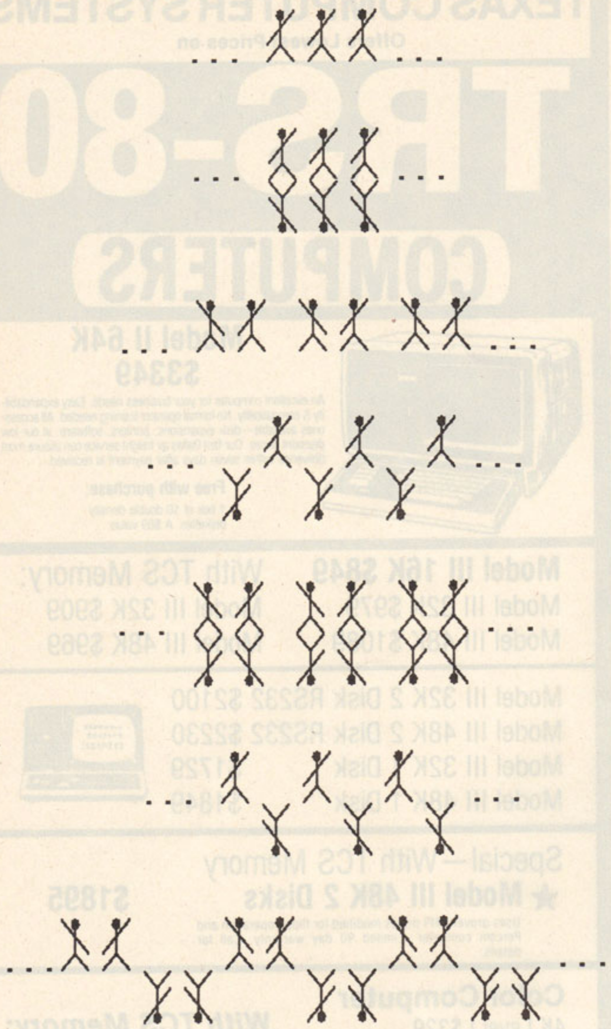


Figure 16: *Frieze patterns of stick men. By combining a basic pattern and its variations, many frieze patterns can be made.*

preserve them. (A symmetry is a one-to-one transformation that preserves distance.) The basic symmetries are translations, rotations, and reflections. We envision the children casting their shadows to form the patterns—or perhaps even lying on the floor. (Of course, mathematical friezes are like lines—infinitely long—but we can enjoy thinking about finite pieces.)

Frieze patterns might look like those in figure 16. However, we will leave the Dance Acts for the frieze patterns as an exercise for the reader.

The Dance Kit can be thought of as a forum for learning introductory programming concepts. A curriculum developer would create a dancer and replacement parts that are of interest to the student; the developer would also select steps, bridges, and initial (sub-)routines that support recommended or assigned exercises. The combination of dancer replacement parts, steps, bridges, and subroutines makes up a "programming language." Alternatively, the student could do more of the creation of the language. The Dance Kit has our approval because of this possibility of dual and flexible use.■

References

1. Bricklin, Dan and Bob Frankston. VisiCalc™ Computer Software Program, 1979.
2. Goldberg, Adele. "Educational Uses of a Dynabook." *Computers in Education*, Volume 3. Great Britain: Pergamon Press Ltd, 1979, pages 247 through 266.
3. Goldberg, Adele and Alan Kay. *Teaching Smalltalk*. Technical Report SSL 77-2, Xerox Palo Alto Research Center, 1977.
4. Goldberg, Adele, Dave Robson, and Dan Ingalls. *Smalltalk-80: The Language and Its Implementation* and *Smalltalk-80: The Interactive Programming Environment* (forthcoming).
5. Gould, Laura and William Finzer. "A Study of TRIP: A Computer System for Animating Time-Rate-Distance Problems." *Proceedings of the IFIP Third World Conference on Computers in Education (WCCE-81)*. July 1981, Lausanne, Switzerland.
6. Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

ToolBox:
A Smalltalk Illustration System

William Bowman and Bob Flegal
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto CA 94304

Computer art is usually thought of as linear, geometric, and repetitious. The Smalltalk group at the Xerox Palo Alto Research Center has been exploring the potential image-making capabilities of the computer-powered display medium for almost ten years. We have investigated the idea of using the computer and its associated display as an art medium for a user/artist to create visual material. We allow the mixture of an artist's freehand sketches with structured commands for manipulating graphic forms. This approach can be contrasted with the more traditional approach where the machine is programmed to "draw," usually with lines, some visual image on the display screen. Figure 1 is a typical computer-generated pattern in which symmetrically ordered lines form an illusion of spherical volume.

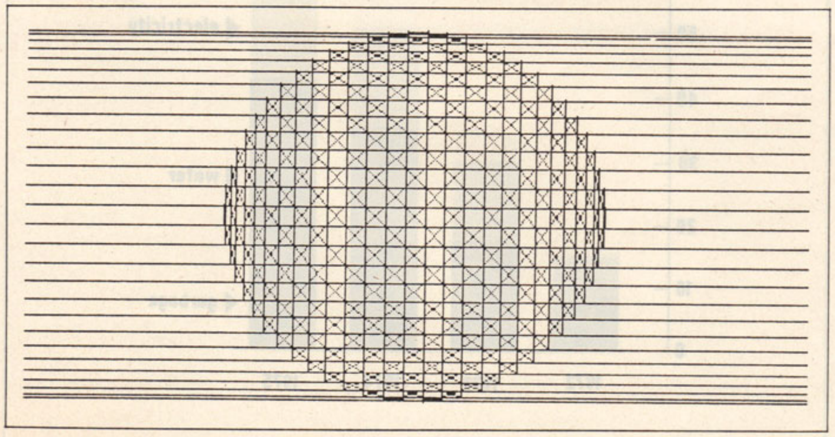


Figure 1: *A typical computer-generated geometric design in which symmetrically ordered lines form an illusion of spherical volume.*

This article reports on one of our developments in the area of computer-assisted image creation. ToolBox is a drawing system designed for general-purpose, interactive image creation and editing. ToolBox was designed jointly by artist William Bowman and computer scientist Bob Flegal to explore graphic specialization within the computer-powered display medium. We were interested in determining the areas within the visual and graphic arts for which the computer-powered display medium is a particularly suitable and efficient graphic tool. To do so we investigated possible tools, techniques, and image-making capabilities of this new medium. The underlying implication (and intention) of this approach is a new role in professional graphics: that of the illustrator/artist who creates images with computer machine tools rather than with conventional hand tools.

ToolBox is a graphics system designed for general-purpose, interactive image creation and editing.

The ToolBox system receives input from the user/artist from a graphics tablet and keyboard and modifies the screen image based on his/her actions. The computer program does not generate the image from a set of programmed drawing instructions. For example, to specify a straight-edge line, the artist need only specify the two end points of the line with the graphics tablet and the program completes the line. This is in contrast to methods where a "pen" is programmed with up, down, and draw commands with coordinates as arguments. This idea is illustrated in figure 2.

System Description

ToolBox consists of a coordinated set of graphic tools that provide a wide range of form construction options for use in testing machine illustration concepts. Five fundamental tool functions comprise the basic graphical form vocabulary. Each of these tools can be modified in its use by one or more of four sets of variables that can affect its form source, color tone, grid spacing, and functional mode. Brief descriptions of the

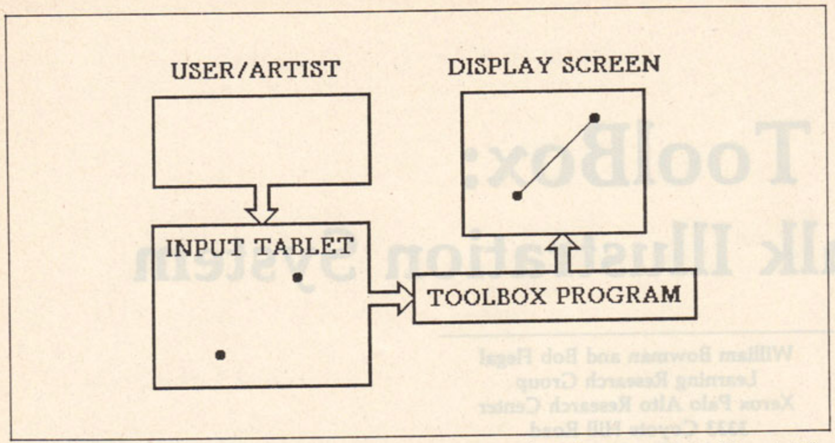


Figure 2: *Modification of the ToolBox screen image based on user/artist input. The computer assists in drawing a line after the artist has entered end points on a graphics tablet.*

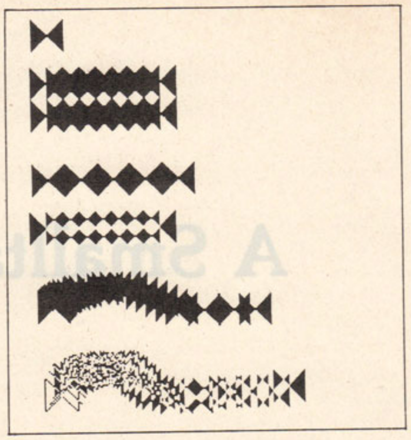


Figure 3: *ToolBox variations on a single form source—two triangles.*

tools and the variables affecting them follow.

Select Form allows the user/artist to select a rectilinear-shaped picture from the display screen as the form source. The form selected can be thought of as a brush which the other five tools use as their source picture. The artist can create a form source using any of the tools available in the ToolBox system, from Smalltalk graphic commands, from the Smalltalk text editor, or from a library of form sources.

The following are ToolBox tool functions:

- •COPY enables continuous individual copies of the form source onto the display screen. This can be thought of as painting on the display screen using a brush (the form source)
- •DRAW enables freehand line drawing or sketching by connecting form source copies with line segments
- •ARC enables curve construction (using spline functions)
- •BLOCK enables solid rectangles to be formed
- •LINE enables straight-line construction between two points

Grid spaces modulate the tools to function on specifiable horizontal and vertical grid lines.

Color tones allow creation of form in black, white, or one of the four intermediate grays (spatial-halftones).

There are three *modes* affecting the copying of source forms onto the display: *store, or,* and *xor*. We called these modes "over," "under," and "reverse."

Art Examples

Figure 3 shows some of the visual effects that are possible with a single form source—two triangles. Since each form source can have five tools, five griddings, six colors, and three modes, the number of possible pictorial effects is staggering.

The display screen upon which the pictures in this article were made is 606 by 808 dots, and each dot is either black or white (no gray scale). The display is refreshed out of the computer's main memory. Thus, to turn a display dot black or white, 1 or 0 is written into memory.

During the programming and development of the ToolBox system, a series of image-making experiments were conducted, both as active input to the evolving design of the system and as a preliminary test of its capabilities. The main purpose of these experiments was to explore the potential of the machine tools for enabling

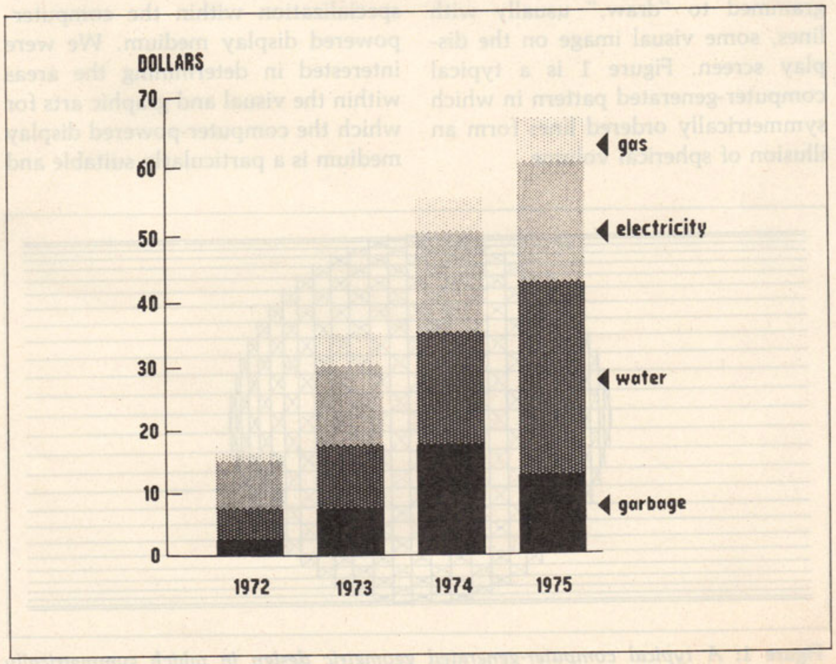


Figure 4: *A bar chart designed by William Bowman.*

Figure 5

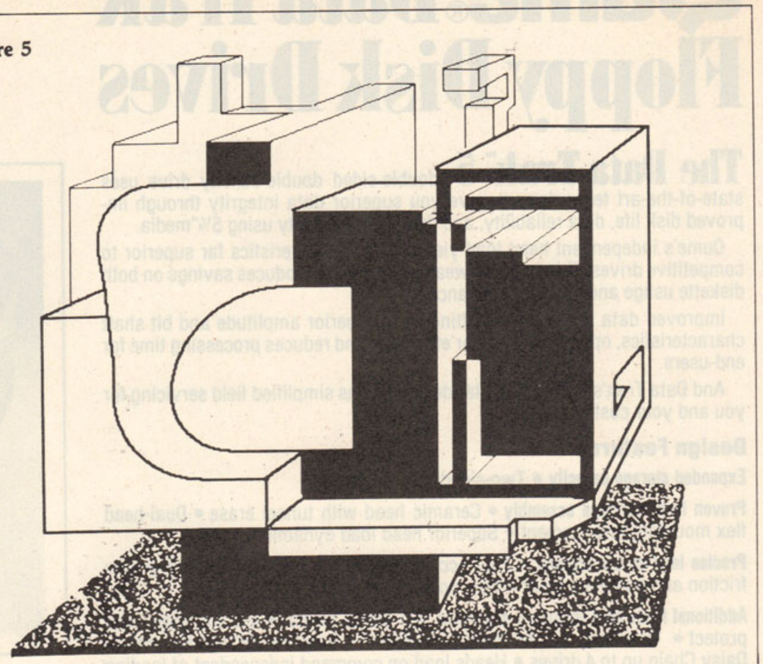
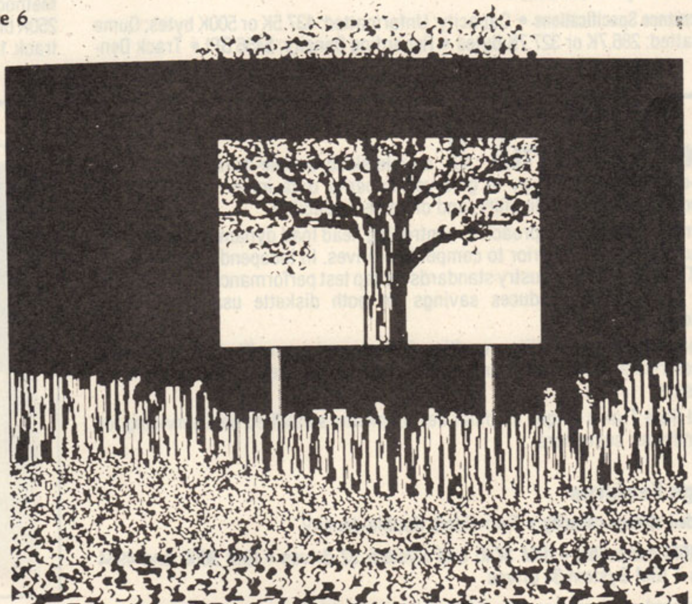


Figure 6



Figures 5, 6, 7, 8: *These four illustrations by Bowman were created using the ToolBox graphics system.*

a variety of graphic strategies for image design and execution. Some of these strategies resembled conventional graphic techniques; however, most of them turned out to be unique to the machine medium.

The remainder of this article consists of pictures that grew out of these early experiments. They are intended as a demonstration of the range and depth of graphic language effects that can be achieved with the ToolBox system.

Figure 4 is a simple bar chart. This figure made heavy use of the BLOCK tool and several grid settings. The

Figure 7

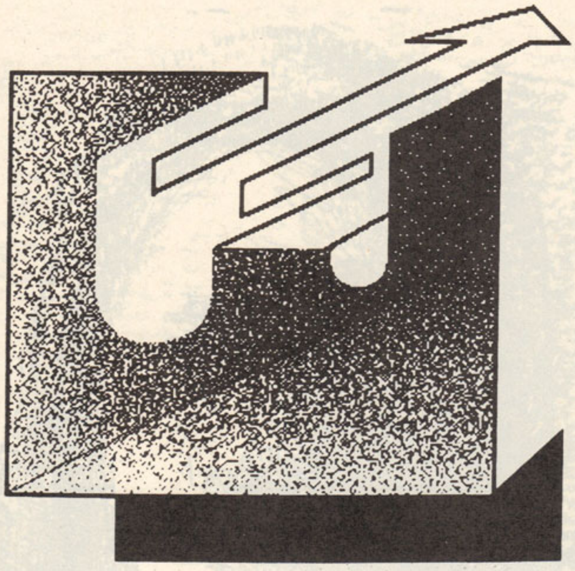


Figure 8

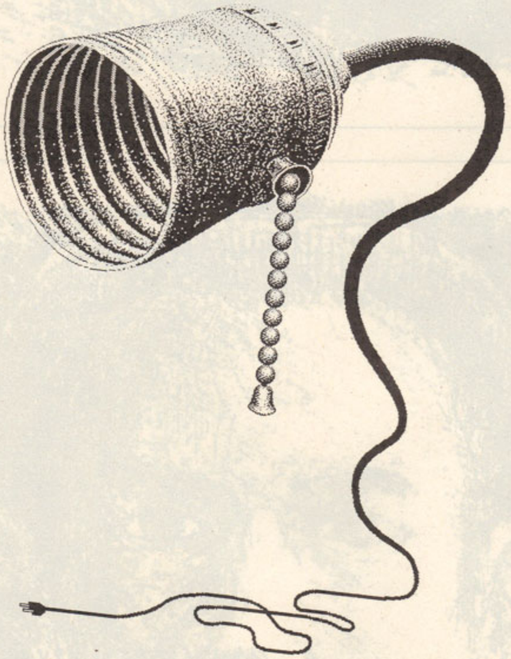


illustration was done in a very short amount of time compared to the amount of time it would take to draw a simple bar chart using conventional media. The ToolBox system is particularly effective for images involving horizontal and vertical elements. (Figures 4, 5, 6, 7, and 8 were executed by William Bowman.)

The BLOCK, LINE, and CURVE tools were heavily used in figure 5. Note the texturing at the bottom of the illustration; it was created using the COPY tool in reverse mode with the color variable set to black.

In figure 6, the COPY tool in

reverse mode was used to create the effects in the bottom half of the image and for the leaves and the bark on the tree.

The perspective effect in figure 7 was easily obtained using the grid settings in the system. The shading was created using a "brush" containing just a few black dots with the COPY tool used in erase mode on a black background.

Note the use of the COPY tool and grid settings to construct the chain on the socket in figure 8. The ToolBox system proved particularly effective for rapid construction of repeated patterns.

The next four pictures were done by Howard Foote, an artist and college art teacher who had never worked on a computer system or terminal. He was contracted to use the system and push it to its graphical limits. In the picture in figure 9, Foote made considerable use of the COPY tool and DRAW tool.

In the picture in figure 10, Foote was able to represent his subject with remarkable loyalty to physical realities when he wanted to and at the same time seemed able to maintain a flexible control over compositional features. His form vocabulary was wide and included a rich use of line, shape, texture, and tonal value.

In the picture in figure 11, Foote made considerable use of the DRAW tool to achieve a free and open effect. Geometrical forms and exact technical mastery of fine detail were the only major areas of pictorial interest with which he did not choose to deal when using the ToolBox system.

The illustration in figure 12 shows a technique often used by Foote: a spatial-saturation strategy using personally constructed form units with the COPY tool in different tones and modes.

The final illustration, figure 13, was done by Bob Flegal. The motifs in the border were taken from North

Figures 9, 10, 11, 12: *The drawings in these four figures were done by Howard Foote, an artist and college art teacher who was contracted to use the ToolBox system and to push it to its graphical limits.*

Figure 9

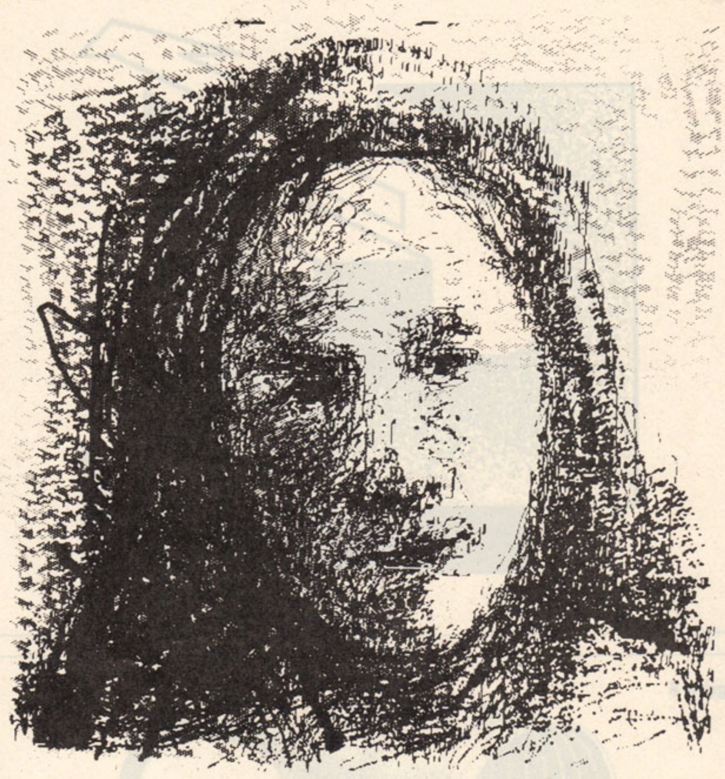


Figure 10

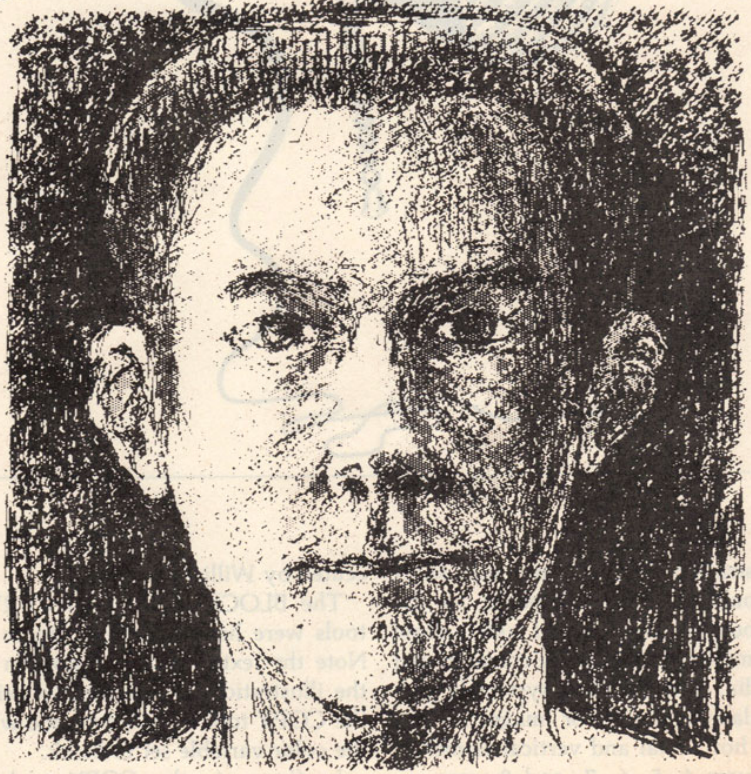


Figure 11

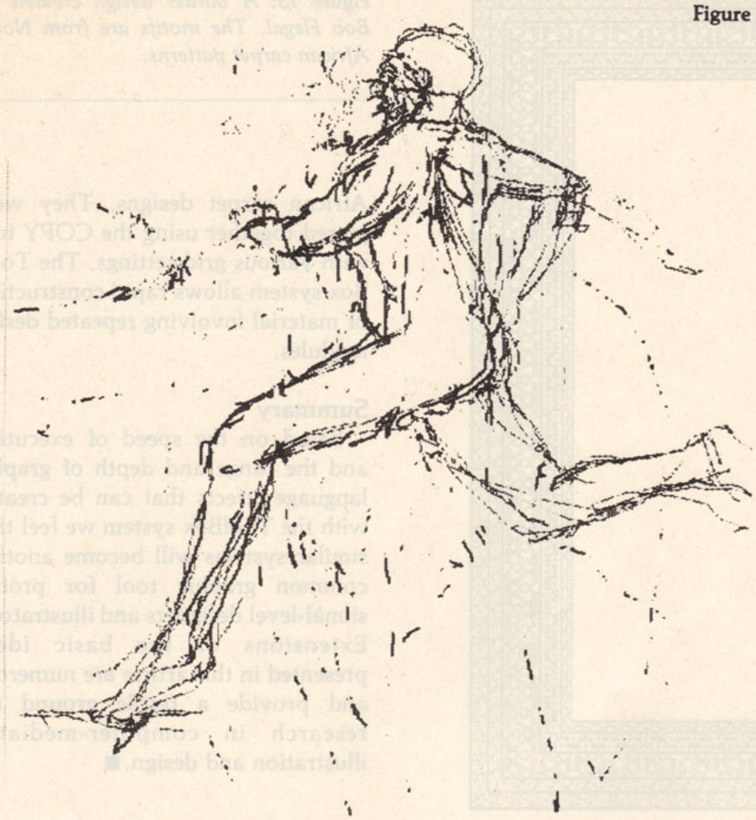


Figure 12

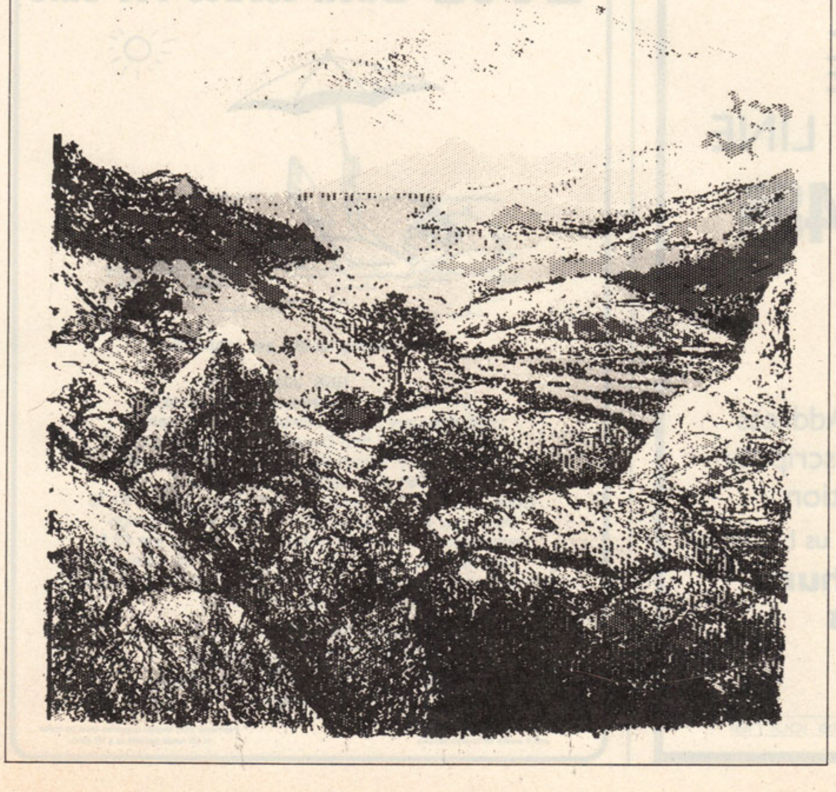
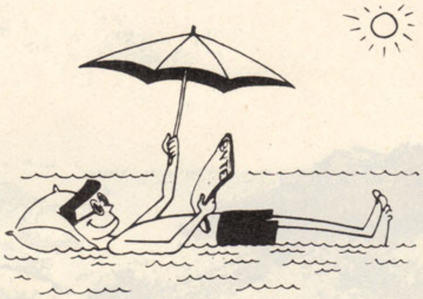


Figure 13: *A border design created by Bob Flegal. The motifs are from North African carpet patterns.*

African carpet designs. They were pieced together using the COPY tool with various grid settings. The ToolBox system allows rapid construction of material involving repeated design modules.

Summary

Based on the speed of execution and the range and depth of graphic language effects that can be created with the ToolBox system we feel that similar systems will become another common graphic tool for professional-level designers and illustrators. Extensions of the basic ideas presented in this article are numerous and provide a fertile ground for research in computer-mediated illustration and design. ■



Virtual Memory for an Object-Oriented Language

Ted Kaehler
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

The amount of information in a person's brain is truly vast; even the amount accessed in the course of a few hours of thought is vast. This is in contrast to the amount of information in the main memory of a computer, which is minuscule by comparison. The exciting thing about computers, though, is that we can use them to extend and enhance our thought. If a computer is to serve effectively as an aid to thought, it must be able to hold enough information to be useful. However, the memory of the largest computer today is so small that it severely limits what that computer can do. There are so many orders of magnitude between the capacity of the brain and the capacity of a computer that given the question "How much memory will the computer need?" the answer should always be "As much as possible."

Software for personal computers is just crossing a threshold of usefulness and flexibility. There are tasks, such as revising a draft of a paper, which are tremendously easier to do with a computer than without. Once you have edited with a computer, it seems absurd to edit by hand. The *number* of tasks for which the computer is essential is growing rapidly, causing a very sharp rise in the demand for storage in each personal computing system. As we design more useful aids to human thought, we will immediately want to access an amount of information closer to the amount in someone's head. Many extraordinary ideas will become software realities in the next few years. And large quantities of memory will be needed to run and store all of that wonderful software.

Given the question "How much memory will the computer need?" the answer should always be "As much as possible."

The practical limit on the size of a computer's memory is cost. Every project, especially a personal computer, has cost limits. The question becomes how to get the most memory for the least cost. Roughly speaking, memory falls into two categories: fast, expensive memory and slow, inexpensive memory. *Main memory* and *core* are common names for the fast, semiconductor memory. The slow memory, *secondary memory*, is almost always a disk. If we bought all slow memory, the processor would continually wait for the disk and would give very poor performance. If we spent all our money on fast memory, we would not get very much of it, and many of the bigger and better programs would not fit in. The game is to buy some fast memory and some slow memory and arrange things so that the processor rarely has to wait for the slow memory. This game, and specifically the mechanism which hides the slow memory from the processor, is called *virtual memory*.

If there were no way at all to predict which byte of memory the processor might want next, it would be impossible to win the game of virtual memory. However, pieces of data that are used together are often stored together, and program instructions tend to be executed and stored in a sequence. The principle of *locality of reference* states that the processor is most likely to access a memory location very near the last

one it accessed (see reference 2 at the end of this article). The game of virtual memory is based on a trick: when the processor starts to ask for bytes from a block of code or data, it should move that code or data into the fast memory. If the processor continues to access that information, all of the accesses will be to fast memory. When the program moves on to a new activity, it may again be forced to get its information from the slow memory. To win the game, a virtual memory must maintain a situation where most of the processor's accesses are to the fast memory. If the strategy fails and the processor often wants data from the slow memory, the entire system will run very slowly.

The act of moving programs or data between the two kinds of memory is called *swapping* (see figure 1). The program that the user is running may or may not control the swapping explicitly. Overlays are large groups of subroutines that are moved to and from the disk under control of the user program. In an *automatic virtual memory*, however, the user program is unaware that swapping is occurring. The programmer does not specify how the program should be divided up into pieces or when swapping should occur.

In certain cases, letting the programmer control swapping directly can result in good performance. However, the virtual memory game is very complex and is played very quickly inside the computer. We believe that the programmer should not be burdened with deciding what part of the data to swap and when to do it. Asking the programmer to instruct the virtual memory is like asking a race car designer to write down, for the driver, exactly how to move the steering wheel in some future race.

In this article, we first look at a common type of automatic virtual memory called *paging*. We then introduce a new type of virtual

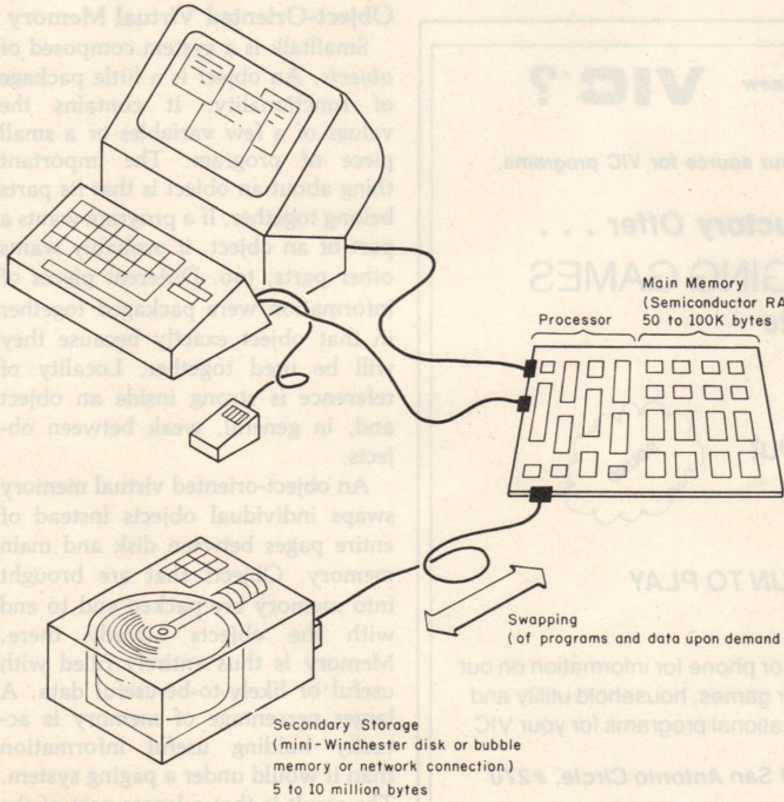


Figure 1: *Main memory, secondary memory, and swapping combine to form a virtual machine that seems to have more memory.*

memory that takes advantage of its knowledge of objects. We describe in detail a specific object-oriented virtual memory for the Smalltalk-76 system and explain how it plays the virtual memory game better than a paging system.

Paging

The most common kind of automatic virtual memory is called paging. In paging, the program is cut up arbitrarily into pieces. Each piece is called a *page* and contains the same number of bytes as every other page —say, 512. There are many more pages than will fit into main memory at once, so most of them stay on the disk. The processor knows only about byte addresses in one large address space called the *virtual address space*. Every time the processor accesses a byte, the address of the byte is checked. The high-order bits of the address tell which page contains that byte. (The low-order bits tell which byte within the page.) If that page is not in main memory, the user program stops. The virtual memory program starts up, finds an old page, moves it to the disk, and brings the desired page into memory in its place. (We will use the term "memory" to refer to the fast, main memory only.) The act of discovering that a page is needed from the disk and bringing it into memory is called a *page fault*.

An advantage of paging is that it works regardless of the contents of the pages. The mechanism needed to determine whether a given page is in memory is simple. Many computers have special hardware to speed up the translation between an address in the virtual space and a page in memory.

There are problems with paging, however. If the program needs a particular byte, the entire page surrounding that byte must be brought into memory. If no other bytes on that page are useful at the moment, a large amount of main memory is wasted. Since programs are cut up arbitrarily into pages in the first place, it is common that the rest of the page has nothing to do with the part currently wanted. Sometimes a significant fraction of memory is taken up by pages from which the processor wants only a few bytes (see figure 2). These pages crowd out pages containing other parts of the program, causing many pages to be swapped to run the rest of the program. The many accesses to slow, secondary memory cause the whole system to be slow.

Another problem with paging is that every address of a byte or a word must be a long address. When an object-oriented language is built on a paging system, a pointer to an object is typically the address of the first word of the object. Every pointer must be capable of reaching any word in the entire virtual space, and each one must have enough bits to span the space. Pointers comprise a large fraction of many programs and data structures. If they could be shorter, more of the program could be packed into one page in memory and the entire program would take fewer pages of memory.

Object-Oriented Virtual Memory

Smalltalk is a system composed of *objects*. An object is a little package of functionality. It contains the values of a few variables or a small piece of program. The important thing about an object is that its parts belong together. If a program wants a part of an object, it probably wants other parts, too. Different pieces of information were packaged together in that object exactly because they will be used together. Locality of reference is strong inside an object and, in general, weak between objects.

An object-oriented virtual memory swaps individual objects instead of entire pages between disk and main memory. Objects that are brought into memory are packed end to end with the objects already there. Memory is thus entirely filled with useful or likely-to-be-useful data. A larger percentage of memory is actually holding useful information than it would under a paging system. The result is that a larger part of the program can fit into memory at once.

There is a penalty for swapping objects, however. Objects are generally smaller than pages, and there are a lot of them in memory at once. The virtual memory program must keep track of which object is in which place in the memory, and it must be able to find out where each object came from on the disk. Managing individual objects is more complicated than managing pages, but the advantage of packing main memory with useful objects makes up for the time spent managing the objects.

By object-oriented virtual memory, we mean a system that swaps objects which have meaning in the high-level language and which are typically small. Segments in the B5500 (reference 4) and objects in HYDRA (reference 5), while being the units of swapping in their systems, are large. These "objects" require tens or hundreds of bytes of overhead information each. An object-oriented virtual memory, in our sense, gives an object the same swapping freedom as a segment and shrinks the overhead to a few bytes per object.

Pointers to Objects

An object consists of *fields*, which hold the values of their named and indexed instance variables. Each field contains a numeric value, which can be interpreted as itself or (usually) as a pointer to another object. This number, called the *object pointer*, is the unique identifier of the other object. Every object has an object pointer. Given an object pointer, the virtual memory must be able to locate that object, whether it is in memory or on the disk (see figure 3).

Creating, destroying, and moving objects in memory is the job of a storage manager. The virtual memory program takes the place of the storage manager (as described in Glenn Krasner's article, "The Smalltalk-80 Virtual Machine," on page 300 of this issue). It fetches and stores the fields of objects, creates new objects, and collects and manages free space. It also keeps track of the length of each object and the Smalltalk class of each object.

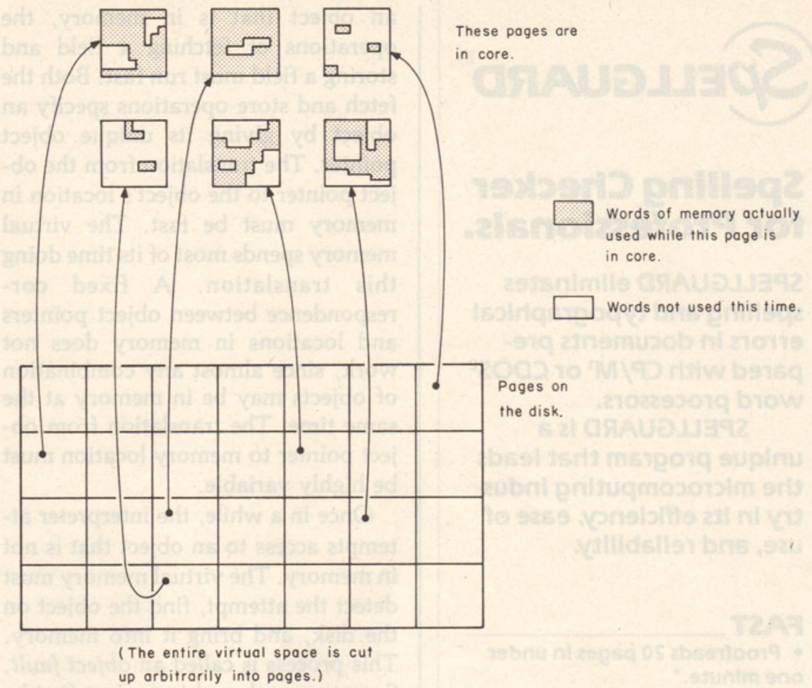

These pages are in core.
Words of memory actually used while this page is in core.
Words not used this time.
Pages on the disk.
(The entire virtual space is cut up arbitrarily into pages.)

Figure 2: *Virtual memory by paging.*

When the interpreter is working on

an object that is in memory, the operations of fetching a field and storing a field must run fast. Both the fetch and store operations specify an object by giving its unique object pointer. The translation from the object pointer to the object's location in memory must be fast. The virtual memory spends most of its time doing this translation. A fixed correspondence between object pointers and locations in memory does not work, since almost any combination of objects may be in memory at the same time. The translation from object pointer to memory location must be highly variable.

Once in a while, the interpreter attempts access to an object that is not in memory. The virtual memory must detect the attempt, find the object on the disk, and bring it into memory. This process is called an *object fault*. Sometimes other objects must first be removed from memory to make room for the incoming object. In order to find an object on the disk, there must be a correspondence between an object pointer and that object's location on the disk. The data needed to hold this correspondence must be compactly represented, as there may be many objects in the system.

OOZE

In 1975 and 1976, Dan Ingalls and I designed and built a virtual memory to support the Smalltalk-74 system, called OOZE (Object-Oriented Zoned Environment). It then became the foundation for the Smalltalk-76 system (reference 3). The combination was very successful, and many interesting projects have been built in it. OOZE serves as an excellent illustration of a usable object-oriented virtual memory implemented entirely in software. At the end of this article, we discuss possible modifications of OOZE for the Smalltalk-80 system.

For OOZE to play the game of virtual memory well, we had to design it to fit the rules. Economics (of our existing hardware) dictated the size of

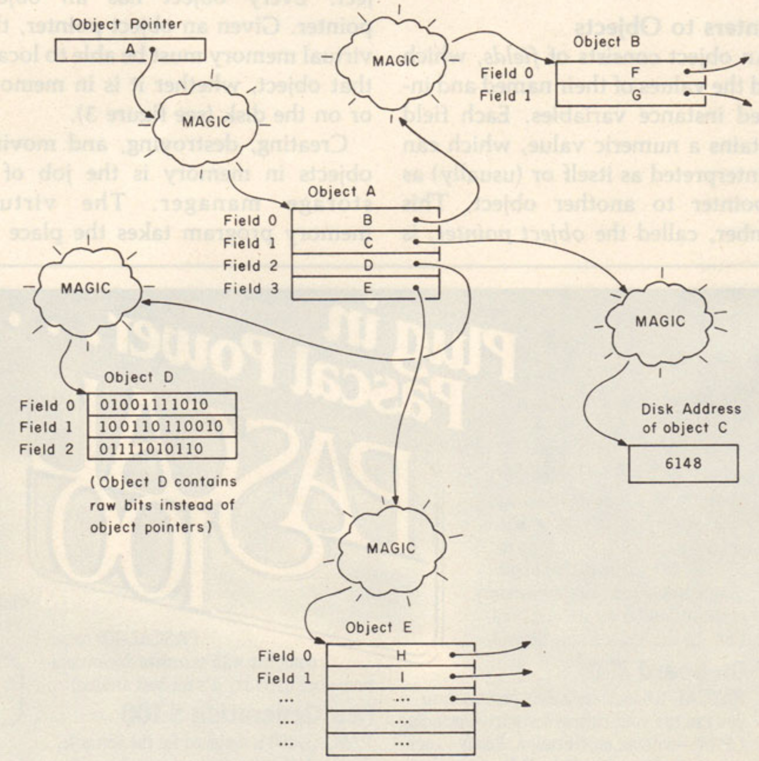


Figure 3: *Objects and object pointers (as seen by casual observers). The "magic" is the unspecified process of translating the value of the object pointer to the actual address at which the object is stored.*

main memory, the size of the disk, and the ratio of their speeds. The rules also included the things that the Smalltalk interpreter expected objects to do. We considered these and decided that in OOZE an object pointer would be 16 bits long, to fit into a machine word. We wanted every combination of 16 bits to be a legal object pointer, giving a total of 64 K objects. With a mean object size of 10 to 20 words, this was a good match to the size of our disk. To guarantee good performance during a fault on an object, we specified that any object can be brought into memory by reading, at most, one place on the disk. We did not allow one disk read to look up the disk address and another disk read to get the actual object.

The design of OOZE centers around the handling of the two important object pointer translations. Finding an object's location in memory from its object pointer must be fast. This mapping must also be flexible, since the exact combination of objects in memory changes from moment to moment. The correspondence between object pointer and memory location is a large hash table, called the *Resident Object Table* (ROT). Of the 64 K objects on the disk, perhaps 4000 are in memory at once. Each of these has an entry in the ROT. To find the location of an object, the hash routine uses the object pointer to compute where to look in the ROT. If it finds an entry whose object pointer matches, that entry also contains the memory address of the object (see figure 4). If the hash routine finds no match in the few entries it searches, the object is not in memory. The magic puffs of smoke in figure 3 depict the act of hashing an object pointer into the ROT to find its memory address.

OOZE must maintain the ROT. When an object is brought in from the disk, OOZE hashes its object pointer and looks in the ROT. When it finds an empty entry among the few possibilities, it claims that entry for the new object. Conversely, when an object is removed from memory and put back on the disk, its entry in the

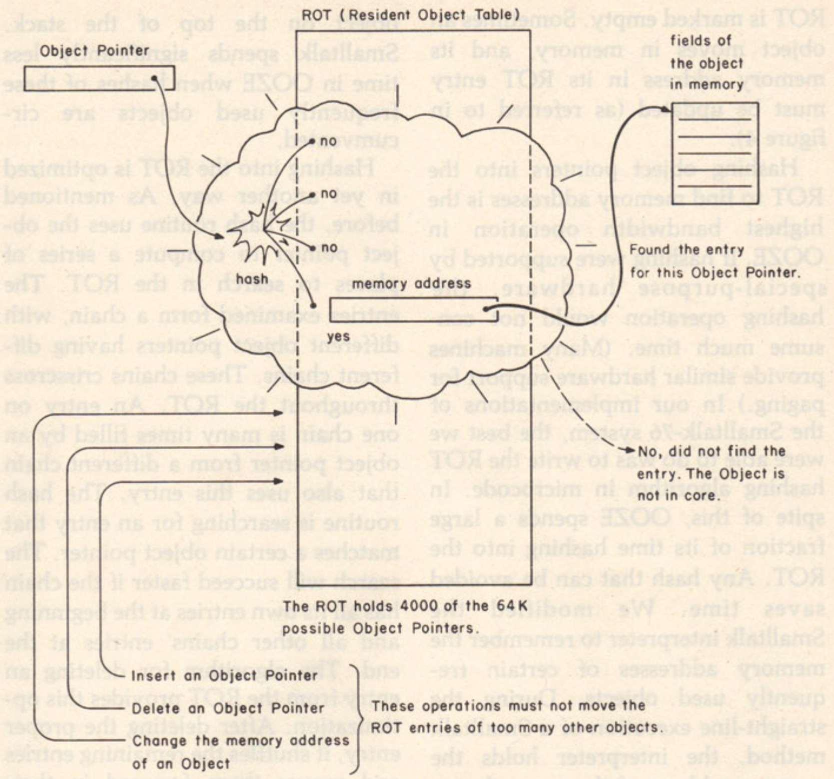

ROT (Resident Object Table)
Object Pointer
fields of the object in memory
no
no
no
hash
memory address
yes
Found the entry for this Object Pointer.
No, did not find the entry. The Object is not in core.
The ROT holds 4000 of the 64 K possible Object Pointers.
Insert an Object Pointer
Delete an Object Pointer
Change the memory address of an Object
These operations must not move the ROT entries of too many other objects.

Figure 4: *Hashing an object pointer in the Resident Object Table (ROT).*

ROT is marked empty. Sometimes an object moves in memory, and its memory address in its ROT entry must be updated (as referred to in figure 4).

Hashing object pointers into the ROT to find memory addresses is the highest bandwidth operation in OOZE. If hashing were supported by special-purpose hardware, the hashing operation would not consume much time. (Many machines provide similar hardware support for paging.) In our implementations of the Smalltalk-76 system, the best we were able to do was to write the ROT hashing algorithm in microcode. In spite of this, OOZE spends a large fraction of its time hashing into the ROT. Any hash that can be avoided saves time. We modified the Smalltalk interpreter to remember the memory addresses of certain frequently used objects. During the straight-line execution of a Smalltalk method, the interpreter holds the memory address of the currently executing method, the receiver, and the object on the top of the stack. Smalltalk spends significantly less time in OOZE when hashes of these frequently used objects are circumvented.

Hashing into the ROT is optimized in yet another way. As mentioned before, the hash routine uses the object pointer to compute a series of places to search in the ROT. The entries examined form a chain, with different object pointers having different chains. These chains crisscross throughout the ROT. An entry on one chain is many times filled by an object pointer from a different chain that also uses this entry. The hash routine is searching for an entry that matches a certain object pointer. The search will succeed faster if the chain has all its own entries at the beginning and all other chains' entries at the end. The algorithm for deleting an entry from the ROT provides this optimization. After deleting the proper entry, it shuffles the remaining entries and moves them forward in their chains. Because of this strategy, the average number of entries examined to find an object in memory is only 1.8. Typically, the resident object table is 80 percent full.

Finding an Object on the Disk

The translation from an object pointer to the disk address of the object is also important. Since a list of the disk addresses of all 64 K objects would easily fill up main memory, OOZE must use a trick. Instead of object pointers being assigned randomly to objects, information is encoded in each object pointer. This is done by dividing the set of object pointers into *pseudoclasses*. The bits in the upper part of the pointer indicate to which pseudoclass that object belongs. All objects in a pseudoclass have the same Smalltalk class and have the same length. The *Pseudoclass Map* is a table that is indexed with the pseudoclass number. There OOZE finds the length of the object and its class (see figure 5). A single Smalltalk class may own as many pseudoclasses as it needs to cover all of its instances. Classes whose instances may have indexable variables, such as class String, own a different pseudoclass for each length or range of lengths. The pseudoclass encoding saves space because each object does not use a word to hold its class or a word to hold its length. Objects in memory in OOZE are actually two words shorter than objects in the Smalltalk-80 system.

The disk address of an object is also found by using its pseudoclass. All objects in a pseudoclass are the same length, and they are stored consecutively on the disk. By knowing which object we want within the pseudoclass, we can compute its offset from the beginning of the pseudoclass. If we know the starting disk address of the pseudoclass, we can add the offset and find the object. The Pseudoclass Map contains the starting disk address of the object's pseudoclass (see figure 5). The low bits of the object's pointer tell which object it is within the pseudoclass. This encoding allows the disk addresses of all 64 K objects to be stored in 512 words of memory.

(There are actually two additional levels of translation for the disk address. Tables for these take another 740 words).

By using the Pseudoclass Map, OOZE can find the disk address of any object from its object pointer. If it is in memory, OOZE also finds the object from its object pointer. *Thus the same object pointer serves to identify and find an object, no matter where the object is.* Because moving an object between disk and memory does not change its pointer, fields that point to the object need not change when the object moves. A field always contains the object pointer of the object to which it refers, regardless of the field's location and regardless of the object's location.

Storage Management

The management of the swapping space has several aspects. Objects are created and destroyed by Smalltalk upon request, and they are also moved in and out of memory. Each of these actions causes insertion or deletion in the ROT and allocation or deallocation in memory. Consider a class that wants to create a new instance: the new instance must receive an object pointer whose pseudoclass is already owned by that class. For this reason, we treat free instances of a class as legitimate objects. They "belong to the class" and can be swapped to and from the disk just like normal instances. Each class keeps a linked list of free instances. The class thinks that there are an infinite number of free instances on the disk, waiting to be swapped in. To create a new instance, the class merely pulls the first object off its "free list." If that object is not in memory, a fault brings it in from the disk. When a free list on the disk runs out, OOZE constructs new free instances on the disk as they are requested.

We have reduced the problem of managing memory and the ROT to the problem of swapping. Main memory has some free blocks between the areas being used for objects. These free blocks are linked together on free lists according to their size. The ROT also contains unused entries, which are marked as such. During an object fault, OOZE

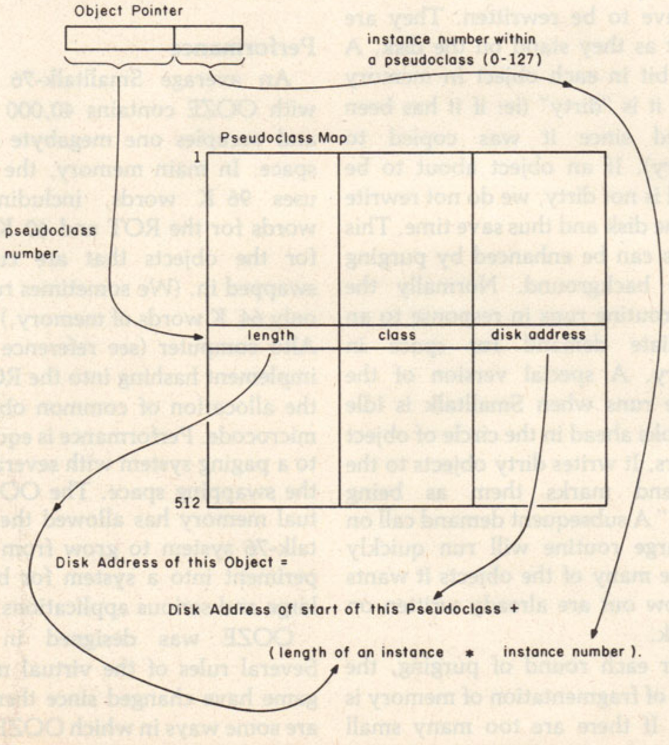

Object Pointer
instance number within
a pseudoclass (0-127)
Pseudoclass Map
1
pseudoclass
number
length
class
disk address
512
Disk Address of this Object =
Disk Address of start of this Pseudoclass +
(length of an instance * instance number).

Figure 5: *Information encoded in an object pointer.*

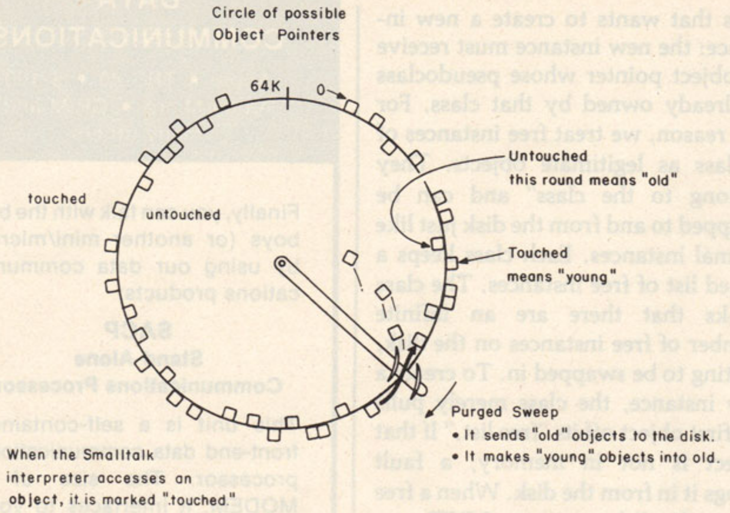


Figure 6: *The order in which objects are purged from main memory.*

claims a free ROT entry and a proper-sized block of memory for the incoming object. Occasionally, OOZE cannot find a legal ROT entry or a free block of memory that is large enough. The fault routine stops, and OOZE starts purging objects from memory by copying them to their proper places on the disk. It then frees the memory space and ROT entries of the objects it throws out. When the purge routine finishes, the fault routine resumes its work.

The purge routine must decide which objects to throw out of memory. To play the game of virtual memory perfectly, OOZE should keep the objects which will be used soon and throw out those which will not. Since OOZE cannot see into the future, it throws out the least recently used objects. Objects that have been active recently are kept, and inactive ones are tossed out. The purge routine examines objects in memory in an order determined by their object pointers. Consider the space of all object pointers to be a circle. The purge routine tours the circle, keeping objects that have been accessed since the last time around. Objects that have remained unaccessed since the routine last visited them are purged to the disk (see figure 6). Typically, it takes several calls on the purge routine to complete a tour of the circle of object pointers.

Purging objects in order of their object pointers has a very important side effect. Since all objects in a pseudoclass are consecutive on the disk, purge sends out the objects it is purging in the same order that they appear on the disk. This minimizes the movement of the disk head and saves time.

Objects that have not been changed since they came in from the disk do not have to be rewritten. They are correct as they stand on the disk. A single bit in each object in memory tells if it is "dirty" (ie: if it has been changed since it was copied to memory). If an object about to be purged is not dirty, we do not rewrite it on the disk and thus save time. This savings can be enhanced by purging in the background. Normally the purge routine runs in response to an immediate demand for space in memory. A special version of the routine runs when Smalltalk is idle and looks ahead in the circle of object pointers. It writes dirty objects to the disk and marks them as being "clean." A subsequent demand call on the purge routine will run quickly because many of the objects it wants to throw out are already written on the disk.

After each round of purging, the degree of fragmentation of memory is tested. If there are too many small blocks and no big ones, we perform a *compaction*. All objects are moved to one end of memory and all free blocks are merged into a single block at the other end. Memory addresses are updated in the ROT entries of the objects that have moved. OOZE performs this operation without using additional storage in order to keep a list of which objects have moved to which place in memory.

As a storage manager, OOZE must detect when an object is no longer being used. Like the storage manager mentioned in Krasner's article, OOZE uses *reference counting*. When the reference count of an object goes to zero, its object pointer is not in any field of any other object. At this point, it is impossible for that object ever to be accessed by the interpreter. OOZE, therefore, puts the object on its class's free list. Before doing so, however, it decreases the reference count of the object pointed to by each field of this object. In the process, more counts may go to zero, and more objects may get freed. To save space, reference counts are only four bits wide. The few objects with fifteen or more fields pointing at them are noted in a separate overflow reference count table.

Performance

An average Smalltalk-76 system with OOZE contains 40,000 objects and occupies one megabyte of disk space. In main memory, the system uses 96 K words, including 8 K words for the ROT and 40 K words for the objects that are currently swapped in. (We sometimes run with only 64 K words of memory.) On the Alto computer (see reference 1), we implement hashing into the ROT and the allocation of common objects in microcode. Performance is equivalent to a paging system with several times the swapping space. The OOZE virtual memory has allowed the Smalltalk-76 system to grow from an experiment into a system for building large and serious applications.

OOZE was designed in 1975. Several rules of the virtual memory game have changed since then. Here are some ways in which OOZE shows its age at Xerox PARC:

- Users can afford more disk and more memory. They want to build systems that contain more than 64 K objects. OOZE cannot be easily expanded beyond this limit.
- Several extensions to the Smalltalk language encourage the user to create lots of classes. OOZE has a limit of 245 classes, and many serious users have encountered this limit.

Naturally, our minds have turned to building a virtual memory for the Smalltalk-80 system with even better performance than OOZE. In 1980, a group of us at Xerox PARC designed just such a system, the *Large Object-Oriented Memory* (LOOM). Individual objects in LOOM carry slightly more overhead than objects in OOZE. (Since users can afford more memory, this is not a problem.) Besides allowing a much larger virtual space and unlimited classes, LOOM provides some new properties:

- LOOM accesses objects that are in memory simply by indexing a table, as does the resident Smalltalk-80 system. LOOM thus saves the time that OOZE spends hashing into the ROT whenever it wants a memory address. During the table lookup which finds an object's memory address, LOOM tests for the case when the object is not actually in memory. To run faster than a hash in OOZE, the test must be very simple and fast.
- LOOM is designed with the idea in mind of grouping objects on the disk. If objects that are faulted on together can be arranged into groups on the disk, the system will run faster. LOOM will be a test bed for schemes that optimize the organization of objects on the disk.

Conclusion

The goal of the virtual memory game is to make a mixture of fast and slow memory perform almost as well as if it were all fast memory. The strategy is to guess what information the processor will need soon and move it to fast memory. In an object-oriented language such as Smalltalk, the object is an excellent unit for locality of reference. Once an object is accessed, it will most likely be accessed again soon. Recently used objects have a similar degree of locality to recently used pages, and many more objects than pages fit into a given amount of fast memory.

OOZE is the first representative of the new category of object-oriented virtual memories. These systems use a construct in the high-level language, the object, as the unit of swapping. Objects as small as one field in length are swapped individually by the same mechanism used for large strings and arrays. To be a member of this category, a virtual memory must also have automatic control of swapping and automatic creation and freeing of objects. While OOZE is implemented in software, we believe that future systems will be implemented like languages: hardware assist for a few high-bandwidth operations, some microcode, and support code in machine- or high-level language. We expect that mature object-oriented virtual memories will identify groups of objects that are used together and swap them as a unit.

As the virtual memory which supports the Smalltalk-76 system, OOZE is interesting in itself. It provides the ability to address 2^N *objects* with *N*-bit pointers. Only currently active objects occupy memory, and they are packed end to end. This provides exceedingly good use of memory. Because the class and length of an object are encoded in the object pointer, that information does not occupy space with each object in memory. Movement of the disk head is reduced because objects are purged to the disk in the order of their disk addresses. OOZE is implemented in software without any special hardware support. It runs in an amazingly sprightly fashion and performs as well as paging systems with several times the swapping space.

The fact that Smalltalk uses objects consistently and completely allows its virtual memory to be radical in design. Object-oriented virtual memories get their power from a close coupling with the high-level languages they serve. The success of OOZE and the changing rules of the virtual memory game have inspired the design of LOOM, a larger and more efficient object-oriented virtual memory.■

References

1. Bell, C G and Alan Newell. *Computer Structures: Readings and Examples*, New York: McGraw-Hill, 2nd edition, 1980.
2. Denning, P J. "Virtual Memory," *Computing Surveys*, Volume 2, Number 3, September 1970, page 153.
3. Ingalls, Daniel H H. "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record, Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978.
4. Shaw, Alan C. *The Logical Design of Operating Systems*, Englewood Cliffs NJ: Prentice-Hall, 1974.
5. Wulf, W A, et al. *HYDRA/C.mmp*, New York: McGraw-Hill, 1981, Chapter 11.