Box 2
Folder 14

Smalltalk press and clippings, 1983-2002

Adele Goldberg papers
Smalltalk
X5774.2010, *Box 2*

102739382

THE COMPUTER HISTORY MUSEUM
1 027 3938 2

CMP
United Business Media

Java • C++ • Ruby • Perl • SQL • JavaScript

#336 MAY 2002

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

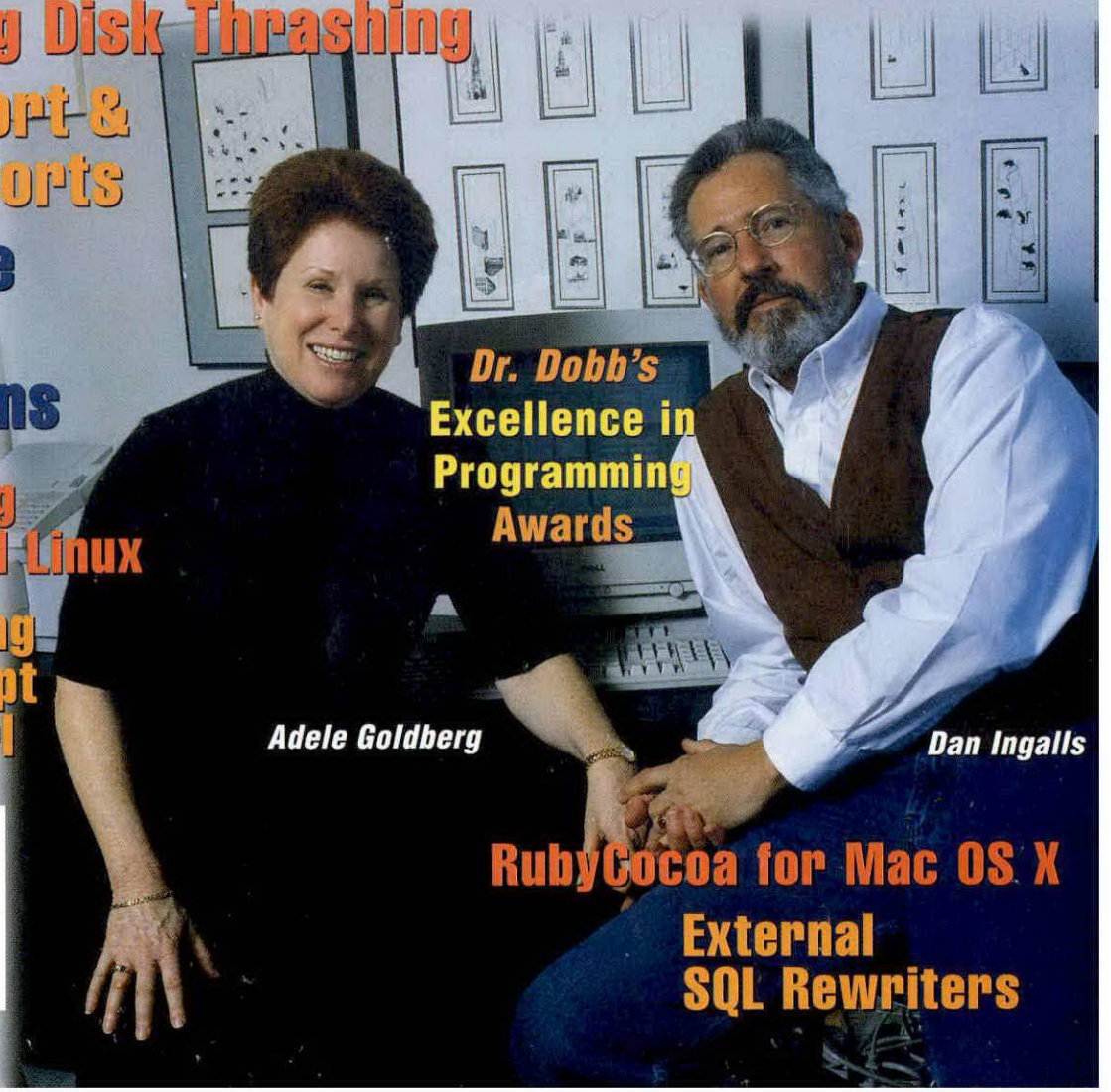
http://www.ddj.com

ALGORITHMS

- Image Scaling with Bresenham
- Avoiding Disk Thrashing
- QuickSort & Radix Sorts
- Multiple Hash Functions

Optimizing Embedded Linux

Generating JavaScript From Perl



Dr. Dobb's Excellence in Programming Awards

Adele Goldberg

Dan Ingalls

RubyCocoa for Mac OS X

External SQL Rewriters

$4.95US $5.95CAN

05>

0 71486 01051 7

CONTENTS

MAY 2002 VOLUME 27, ISSUE 5

FEATURES

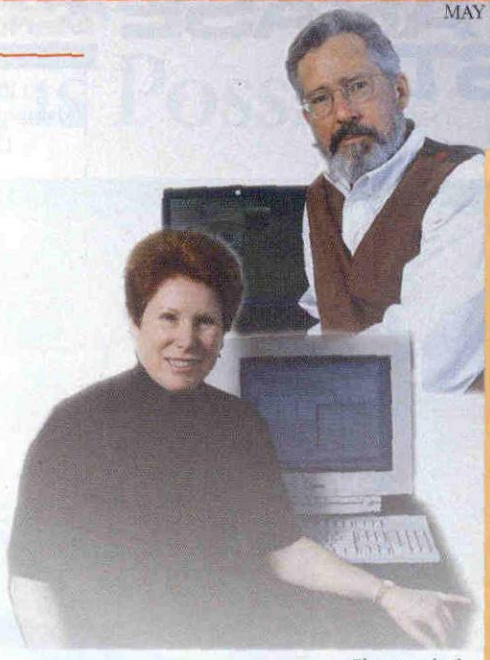
EMBEDDED SYSTEMS

INTERNET PROGRAMMING

COLUMNS

TECHNETCAST
http://www.ddj.com/technetcast/
O'REILLY 2002 BIOINFORMATICS CONFERENCE
LINCOLN STEIN ON BIOINFORMATICS
TERRY GAASTERLAND ON INTEGRATING GENE EXPRESSION & GENOME SEQUENCE DATA

WORKBENCH



Photography by Pat Johnson Studios.

FORUM

RESOURCE CENTER

As a service to our readers, source code and related files, and author guidelines are available at *http://www.ddj.com/*. Source code is also available via anonymous FTP from ftp.ddj.com (199.125.85.76). Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@ddj.com, faxed to 650-513-4618, or mailed to *Dr. Dobb's Journal*, 2800 Campus Drive, San Mateo CA 94403.

For subscription questions, call 800-456-1215 (U.S. or Canada). For all other countries, call 850-682-7644 or fax 303-661-1181. E-mail subscription questions to ddj@neodata.com or write to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188.

Back issues may be purchased for $9.00 per copy (which includes shipping and handling). For issue availability, send e-mail to orders@cmp.com, fax to 785-841-2624, or call 800-444-4881 (U.S. and Canada) or 785-838-7500 (all other countries). Back issue orders must be prepaid. Please send payment to *Dr. Dobb's Journal*, 1601 W. 23rd Street, Suite 200, Lawrence, KS 66046-2700.

Individual back articles may be purchased electronically at http://www.ddj.com/ as ZIP archives.

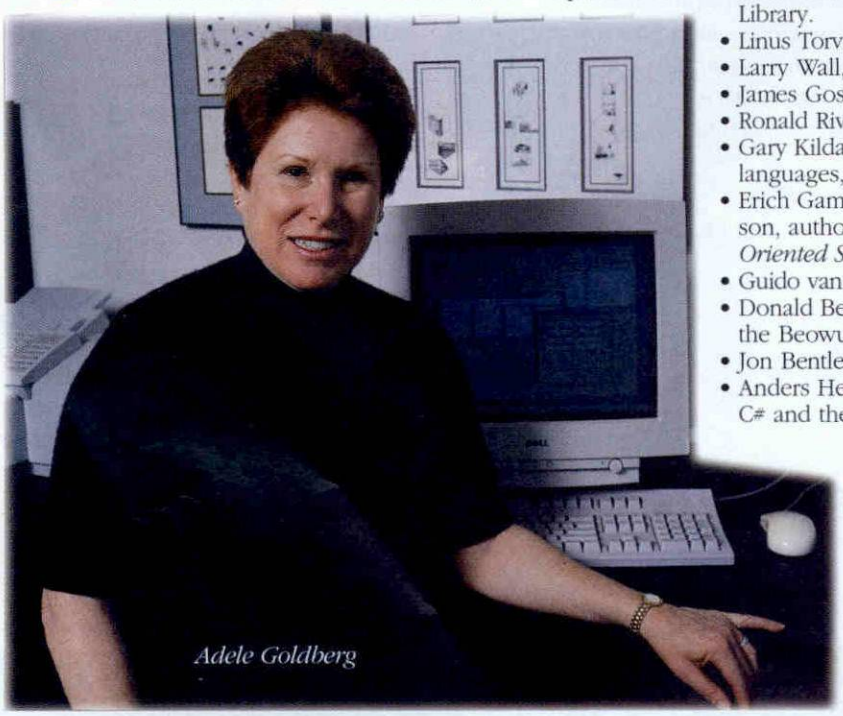
NEXT MONTH: Communication and networking is our theme in June.

DR. DOBB'S JOURNAL (ISSN 1044-789X) is published monthly by CMP Media LLC., 600 Harrison Street, San Francisco, CA 94017; 415-905-2200. Periodicals Postage Paid at San Francisco and at additional mailing offices. SUBSCRIPTION: $34.95 for 1 year; $69.90 for 2 years. International orders must be prepaid. Payment may be made via Mastercard, Visa, or American Express; or via U.S. funds drawn on a U.S. bank. Canada and Mexico: $45.00 per year. All other foreign: $70.00 per year. U.K. subscribers contact Jill Sutcliffe at Parkway Gordon 01-49-1875-386. POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80328-6188. GST (Canada) #R124771239. Canada Post International Publications Mail Product (Canadian Distribution) Sales Agreement No. 0548677. FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 30 Montgomery St., Jersey City, NJ 07302, 212-332-7100. Entire contents © 2002 by CMP Media LLC., unless otherwise noted on specific articles. All rights reserved.

2002 Dr. Dobb's Excellence in Programming Awards

Since 1995, *Dr. Dobb's Journal* has presented its Excellence in Programming Award to individuals who, in the spirit of innovation and cooperation, have made significant contributions to the advancement of software development. Past recipients of the Dr. Dobb's Excellence in Programming Award include:

- Alexander Stepanov, developer of the C++ Standard Template Library.
- Linus Torvalds, for launching Linux.
- Larry Wall, author of Perl.
- James Gosling, chief architect of Java.
- Ronald Rivest, educator, author, and cryptographer.
- Gary Kildall, for his work in operating systems, programming languages, and user interfaces.
- Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson, authors of *Design Patterns: Elements of Reusable Object-Oriented Software*.
- Guido van Rossum, Python creator.
- Donald Becker, for his contributions to Linux networking and the Beowulf Project.
- Jon Bentley, computer-science author and researcher.
- Anders Hejlsberg, developer of Turbo Pascal and architect of C# and the .NET Framework.


Adele Goldberg

The recipients of this year's award, Adele Goldberg and Dan Ingalls, are pioneers in the area of object-oriented programming in general, and the Smalltalk language and development environment in particular. As researchers at Xerox's Palo Alto Research Center (PARC), Goldberg and Ingalls each recognized in their own way the promise of objects, and they were in a unique position to put those theories into practice in an architecture based on objects at every level.

Although we take objects for granted today, these two researchers helped to bring object-oriented programming into the real world for the first time almost 30 years ago, from the highest level of users and their information modeling needs to the lowest levels of syntax, compilation, and efficient message passing.

Looking back on the original work at Xerox, Goldberg later said it tackled one of the most difficult and problem-prone steps in software development—identifying terms and relationships as understood by human participants of a particular situation with those understood by a computer.

To that end, Goldberg believed that:

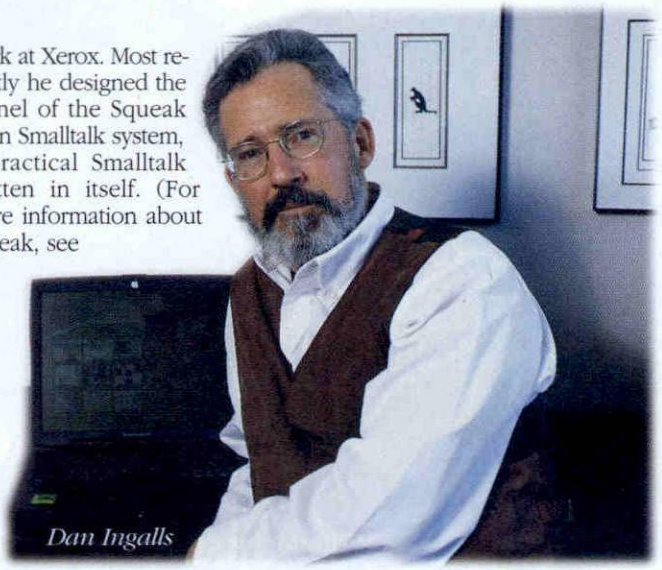
- Interactive, incremental software-development environments could produce a qualitative improvement in software-development productivity.
- Software could be designed in autonomous reusable units, each corresponding to identifiable entities (conceptual as well as physical) in the problem domain that communicate through well-defined interfaces.
- The model, or framework, for how these units work together represents both a process and vocabulary for talking about the problem domain.
- We should think about writing software in the context of building systems, rather than in the context of black box applications.

As early as 1977, Goldberg, along with Alan Kay, presented the goals for the Smalltalk research efforts in a paper entitled "Personal Dynamic Media" (*IEEE Computer*, March 1977). She went on to author and coauthor many of the definitive books on Smalltalk-80 programming including, with David Robson, the seminal *Smalltalk-80: The Language and Its Implementation* (Addison-Wesley, 1989, ISBN 0201136880), as well as numerous papers on object technology. Goldberg edited *The History of Personal Workstations* (ACM/Addison-Wesley, 1988; ISBN 0201112590); coedited with Margaret Burnett and Ted Lewis *Visual Object-Oriented Programming* (Prentice Hall, 1995; ISBN 0131723979); and coauthored with Kenneth Rubin *Succeeding with Objects: Decision Frameworks for Project Management* (Addison-Wesley, 1995; ISBN 0201628783).

Goldberg received her Ph.D. in Information Science from the University of Chicago for work carried out jointly at Stanford University. She also holds an honorary doctorate from the Open University (UK) in recognition of contributions to computer science education. After more than a decade as a researcher and laboratory manager at Xerox PARC, Goldberg became the founding CEO of ParcPlace Systems, the PARC spin-off that developed commercially available object-oriented application-development environments. Goldberg currently is founder of Neometron, a consulting company that focuses on dynamic knowledge management and support for project-based online communities.

From 1984 to 1986, Goldberg was president of the ACM, recipient of the 1987 ACM Systems Software Award along with Dan Ingalls and Alan Kay, and is an ACM Fellow. She received *PC Magazine*'s Lifetime Achievement Award in 1990.

Like Goldberg, Dan Ingalls was an original member of the PARC team that developed Smalltalk. He has been the principal architect of numerous Smalltalk virtual machines and kernel systems. The first of these, Smalltalk-72, supported the work reported in "Personal Dynamic Media." Smalltalk-76, described in ACM's 1978 Principles of Programming Languages (POPL) proceedings (and available at http://users.ipa.net/~dwighth/smalltalk/St76/Smalltalk76ProgrammingSystem.html), was the first modern Smalltalk implementation with message syntax, compact compiled code, inheritance and efficient message execution, and its architecture endures in Smalltalk-80, the major documented release of Smalltalk work at Xerox. Most recently he designed the kernel of the Squeak open Smalltalk system, a practical Smalltalk written in itself. (For more information about Squeak, see ftp://st.cs.uiuc.edu/Smalltalk/Squeak/docs/OOPSLA.Squeak.html.) Ingalls also invented the BitBlt graphics primitive and pop-up menus, and was the principal designer of the Fabrik visual-programming environment while at Apple Computer.


Dan Ingalls

Photography by Pat Johnson Studios.

Ingalls received his Bachelor's degree in physics from Harvard University, and Masters in electrical engineering from Stanford University. He is a recipient of the ACM Grace Hopper Award and the ACM Software Systems Award. Ingalls currently works with Alan Kay and other seasoned Smalltalkers at Viewpoints Research Inc., where he is working to complete an architecture for modular Squeak content that is sharable over the Internet. He supports an active Squeak community (http://www.squeak.org/) through his participation in e-mail discussions, attention to periodic releases, and other support at all levels. He also runs Weather Dimensions (http://www.WeatherDimensions.com/), a company that sells a weather station he designed.

Although Goldberg and Ingalls worked at very different levels, the breadth of their collaborative territory is what shaped the final result. Ingalls says of his technical achievements, "I loved the challenge in efficiency and generality that it took to make Smalltalk real, but what gives me the most satisfaction looking back is that we built a serious system that is actually fun to use. We had a passion, inspired by Alan, to liberate the beauty of computer science from the barnacled past of ad hoc engineering." Goldberg adds, "During the PARC days, the opportunity to work with children and other nontechnical users kept us focused on how to use rigorously what people already know informally about objects. But the most thrilling experience for me was to work with ParcPlace customers in both large and small companies, and see how our technology enabled them to finally break the barrier between business understanding and systems implementation."

At Adele Goldberg's request and in her name, *Dr. Dobb's Journal* is pleased to make a grant of $1000 to the Girl's Middle School (http://www.girlsms.org/), a San Francisco Bay Area all-girls middle school that focuses on math and technology. At Dan Ingalls request and in his name, we are happy to make a $1000 grant to the The Sierra Nevada Children's Museum in Truckee, California. Please join us in honoring Adele Goldberg and Dan Ingalls who once again remind us that a mix of technology, innovation, vision, and cooperative spirit is fundamental to advancement in software development.

DDJ

Debug Java™ and C++ code side by side with Forte™ tools

```
/*
 * SortThread.cc
 *
 */

void *(*sort_func[SORT_CT])(void*) =
{bubblesort, binary_insertion_sort, heapsort, quicksort_stub,
 shakersort, straight_insertion_sort, straight_selection_sort};

JNIEXPORT void JNICALL Java_SortThread_sort_1num
(JNIEnv * env, jclass cl, jint sort_no) {
    if (sort_status[sort_no-1] == SORT_NONE)
        sort_status[sort_no-1] = SORT_SELECTED;

    if (sort_status[sort_no-1] == SORT_SELECTED) {
        int sen = SORT_ELEMENT_NO;
        setup_sort_table((sort_no-1), sen);

    }
}

/*
 * setup_sort_table()
 *
 */
static void setup_sort_table(int sort_type, int elements) {
    int i, j;
}
```

```
/*
 * SortThread.java
 *
 */

import java.lang.*;

public class SortThread extends javax.swing.JFrame {
    final static int bubblesort = 1;
    final static int binary_insertion_sort =2;
    final static int heapsort = 3;
    final static int quicksort = 4;
    final static int shakersort = 5;
    final static int straight_insertion_sort = 6;
    final static int straight_selection_sort = 7;

    public native void sort_proc();
    public native static void sort_num(int i);
    public native static void sort_reset();

    /** Creates new form SortThread */
    public SortThread() {
        super("SortThread");
        initComponents();
    }
```

Before:	**Insert your print statements; Fire-up the Java debugger...**
With Forte Tools:	**Step!**

Forte Development tools from Sun

Whether you are developing in Java,™ C++, C or the Fortran programming language, Sun's Forte™ tools provide a complete, end-to-end solution for developing entry- to enterprise-class applications. And now you can use a single tool to efficiently debug Java classes and legacy programs transparently—eliminating the need to debug your applications with multiple tools from different vendors. This can save you a great deal of time and effort, and increases the quality of your code. You can also use the Native Connector Tool, which automatically creates bindings between Java objects and C or C++ libraries. For a flash demonstration, go to **www.sun.com/forte**

Sun microsystems

©2002 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, Java and Forte are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

PROGRAMMING BY REHEARSAL

BY WILLIAM FINZER AND LAURA GOULD

An environment for developing educational software

PROGRAMMING BY REHEARSAL is a visual programming environment that nonprogrammers can use to create educational software. It combines many of the qualities of computer-based design environments with the full power of a programming language. The emphasis in this graphical environment is on programming visually; only things that can be seen can be manipulated. The design and programming process consists of moving "performers" around on "stages" and teaching them how to interact by sending "cues" to one another. The system relies almost completely on interactive graphics and allows designers to react immediately to their emerging products by showing them, at all stages of development, exactly what their potential users will see.

The process is quick, easy, and enjoyable; a simple program may be constructed in less than half an hour. The beginning set of 18 primitive performers, each of which responds to about 70 cues, can be extended as the designers create new composite performers and teach them new cues.

We were motivated to undertake this project by our desire to give programming power to those who understand how people learn; we wanted to eliminate the need for programmers in the design of educational software. Programming by Rehearsal is implemented in the Smalltalk-80 programming environment and runs on a large, fast, personal machine: the Xerox 1132 Scientific Information Processor (the Dorado).

COMPUTERS AND INTUITION

In the spring of 1980 our attention was focused on a topic we called Computers and Intuition. It seemed to us that newly available, high-resolution computer images, combined with interactive control over these images, constituted a new medium for the presentation of information and concepts. We were particularly concerned with the implications that this interactive computer graphics medium might have for education.

We were also thinking about how paradoxical it was that the computer was often viewed as an engine for improving cognitive and analytical skills, while it might turn out that because of its superlative dynamic graphics, its main new contribution to education might be in the enhancement of nonanalytical, intuitive thought.

Such ideas were certainly not new. Even 15 years ago, a few farseeing people proposed that computer graphics would have a profound effect on human learning. As Brown and Lewis wrote in 1968, "In the same way that books support man's linear and verbal thinking, machines will support his graphic and intuitive thought processes." (See reference 1.) Similarly, in 1969 Tony Oettinger wrote "Computers are capable of profoundly affecting science by stretching human reason and intuition, much as telescopes or microscopes extend human vision." (See reference 2.) It seemed that now we had both the software and hardware to realize these visions.

From these ruminations grew the design and implementation of a system called TRIP, which attempted to give students an intuitive understanding of algebra word problems through the manipulation of high-resolution pictures. (See reference 3.) TRIP, implemented in the Smalltalk-76 system (see reference 4) on research hardware, a Xerox Alto, took about two months to design and four months to implement. It was structured in the form of a kit so that

(*text continued on page* 188)

William Finzer is a consultant with the System Concepts Laboratory at the Xerox Palo Alto Research Center and an instructor and curriculum developer in the mathematics department at San Francisco State University (1600 Holloway, San Francisco, CA 94132).

Laura Gould has been a member of the Smalltalk group at the Xerox Palo Alto Research Center for the past seven years. She is now National Secretary of Computer Professionals for Social Responsibility (POB 717, Palo Alto, CA 94301).

In the Rehearsal World, only things that can be seen can be manipulated

(*text continued from page* 187)
teachers could add new time-rate-distance problems fairly easily; it included a diagram checker, an animation package, an expression evaluator, and an extensive help system. Members of the computing profession were impressed that we were able to bring to life such a complex, general, graphical, yet robust and helpful system in such a short time. Educators, however, were usually aghast that so much time and effort were needed to produce a single system and that the result was, in their view, so limited.

After we had pilot-tested TRIP and were thinking about what project to take on next, we realized that our interest had shifted up one level, from the actual design of educational software to the design of a "design environment" for educators. As our colleagues were busy building the Smalltalk-80 environment (see references 5, 6, 7, and 8), we undertook the task of extending and reifying that environment to allow curriculum designers who did not program to implement their own creative ideas.

Designer Control

The work described here is based on the belief that it should be possible to place the control of interactive computer graphics in the hands of creative curriculum designers, those with an understanding of the power of such systems but not necessarily with the ability or willingness to write the complex programs that are necessary to control the systems.

Design and implementation constitute two phases of a feedback loop. In most design situations, in which programming is a separate and specialized skill, the designer must somehow convey embryonic ideas to a programmer, perhaps by sketching on paper or talking. Then the programmer goes away to write a program so that something shows on the screen to which the designer can respond. This process introduces interruption, distortion, and delay of creative design.

In the creation of educational software it is particularly important that the design decisions be made by someone who understands how students learn and what they enjoy rather than by someone whose expertise is in how computers work. Too much of the educational software we see today has a lot of fancy graphics but little real learning content. We hope that if educators have more direct control of the computer, they will create high-quality software.

In the environment we describe here, the designer begins by sketching the description, not in words or on paper, but directly on the computer screen. This sketching is not free-form but is done with the aid of specially provided graphical entities. If the designer's ideas are rather vague, the process of sketching may help to define them; if the ideas are well defined, they can be quickly accepted, rejected, or improved. In either case, nothing is lost in the translation process, as the only intermediary between the designer and the product is a helpful, graphical computer system that gives immediate response. Since there is no waiting, the designer is involved in a collaborative, creative process in which there is minimal investment in the current production; thus a poor production can be rejected quickly and easily, and a good one pursued and improved.

The Rehearsal Metaphor

A large, supportive design environment needs a potent metaphor in which the unfamiliar concepts of programming will have familiar, real-world referents. Our goal was that the metaphor would serve as a guide to the designers without getting in their way.

Smalltalk is an object-oriented language. This means that all the basic elements of programming—strings, numbers, complex data structures, control structures, and procedures themselves—are treated as objects. Objects interact with other objects by sending messages. Logo is an example of a programming language with one object, a Turtle, which can be sent a limited number of messages such as FORWARD 20. Smalltalk has many kinds of objects that respond to a wide variety of messages.

Our immersion in Smalltalk led us to extend the object-message metaphor to a theater metaphor in which the basic components of a production are performers; these performers interact with one another on a stage by sending cues. We call the design environment the Rehearsal World and the process of creating a production Programming by Rehearsal.

Everything in the Rehearsal World is visible; there are no abstractions and only things that can be seen can be manipulated. Almost all of the designer's interactions with the Rehearsal World are through the selection (with a mouse) of some performer or of some cue to a performer. Assuming that a designer has the germ of an idea, the creation of a Rehearsal World production involves:

- Auditioning the available performers by selecting their cues and observing their responses to determine which are appropriate for the planned production. If a production involves getting the student to write stories using pictures, the designer might choose a text performer and a picture performer because the former responds to the cues *setText:* and *readFromKeyboard* and the latter responds to *growBy:* and *followTheMouse*.
- Copying the chosen performers and placing them on a stage.
- Blocking the production by resizing and moving the performers until they are the desired size and in the desired place.
- Rehearsing the production by showing each performer what actions it should take in response either to student (user) input or to cues sent by other performers.
- Storing the production away for later retrieval.

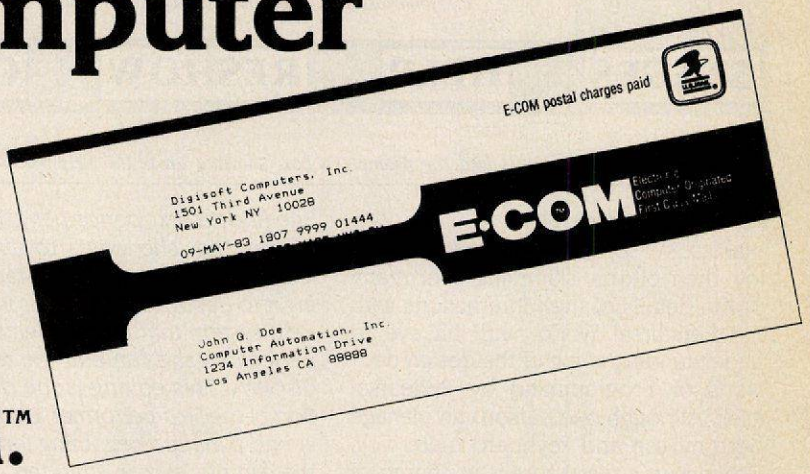
A Scenario

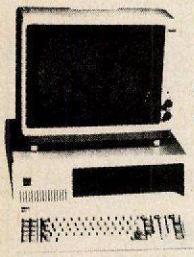
Static words and pictures on paper are a poor substitute for direct experience with a dynamic, interactive, computer design environment. Nevertheless, we shall try to give the flavor of what it is like to use the Rehearsal World through a simple scenario involving two novice designers, Laura and Bill. Suppose that these designers are interested in language curriculum and would like to

(*text continued on page* 190)

Send 2000 Letters Per Hour via Your Personal Computer

Delivered in 48 hours or sooner at 26 cents a piece using MAIL-COM.™





Presenting E-Com

Two years ago the U.S. Postal Service quietly announced the E-Com® Service, enabling specially equipped personal computer users to bypass costly manual mail preparation, by electronically submitting their messages and mailing lists directly to the Postal Service via modem.

This high speed computer originated mail arrives at its destination within 48 hours—often less—in an attention-grabbing blue E-Com envelope.

Announcing MAIL-COM. Only from Digisoft Computers.

MAIL-COM is powerful software you can use with your personal computer to access E-Com. With your personal computer, a modem and MAIL-COM you can send from 200 to 2000 letters per hour for just 26¢ each. Typed, addressed, folded, inserted, sealed and delivered. Complete.

MAIL-COM is the complete integrated software available for E-Com operation. It's easy to use. No special training is necessary. And since Digisoft Computers developed MAIL-COM in accordance with U.S. Postal Service specifications, users are guaranteed certification for use upon purchase of MAIL-COM software.

MAIL-COM is the easiest and most economical way to do your mailings.

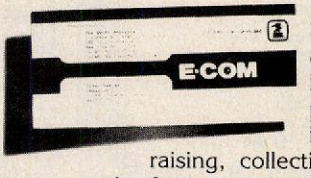
MAIL-COM includes a complete letter editor and address maintenance program, as well as communications software.

Directly interfaces with dBASE II, Wordstar, MailMerge and other databases.

Each letter in your mailing can be identical or all can contain variable insertions. MAIL-COM operates all the features offered by E-Com.

Thousands of Uses.

If you have need for fast, economical mass mailing capabilities, MAIL-COM puts you and E-Com together.



Use it for new product announcements, invitations to press events, invoicing, fund raising, collection, bulletins to your sales force, new business prospecting, reactivation of customers and much, much more. Every department in your company will have use for MAIL-COM.

Don't Delay

With MAIL-COM you could be saving time and money on fast, efficient E-Com letters. MAIL-COM software is available for the IBM PC, PCJr., Kaypro, CP/M, Apple II and other formats. Order today. Call 212-734-3875.

Digisoft

Digisoft Computers, Inc.

(212) 734-3875

Circle 105 on inquiry card.

Retail Dealer Inquiries Invited

Digisoft Computers Inc.
Attn: MAIL-COM Marketing
1501 Third Avenue
New York, NY 10028

☐ Yes! I want to eliminate the 6 costliest steps in preparing my organization's business mail. Please RUSH my MAIL-COM software to me immediately.

I'll need software for:

☐ IBM PC ($195) ☐ CP/M ($195) (specify disk format)
☐ Victor ($195)
☐ Alpha Micro ($495) ☐ Other (specify) _____
☐ Apple II ($195.00)

☐ My check or money order is enclosed (residents of New York State add sales tax).

☐ Charge my ☐ Visa or ☐ MasterCard:

Account No._____ Exp. Date_____

Name_____

Address_____

City_____

State_____ Zip_____

Telephone(_____)_____

© 1983, Digisoft Computers, Inc.

The E-COM® Service is a registered trademark of the U.S. Postal Service. MailMerge and WordStar are registered trademarks of Micropro International. dBase II is a registered trademark of Ashton-Tate, Inc. IBM PC and PCJr. are registered trademarks of IBM Corp.

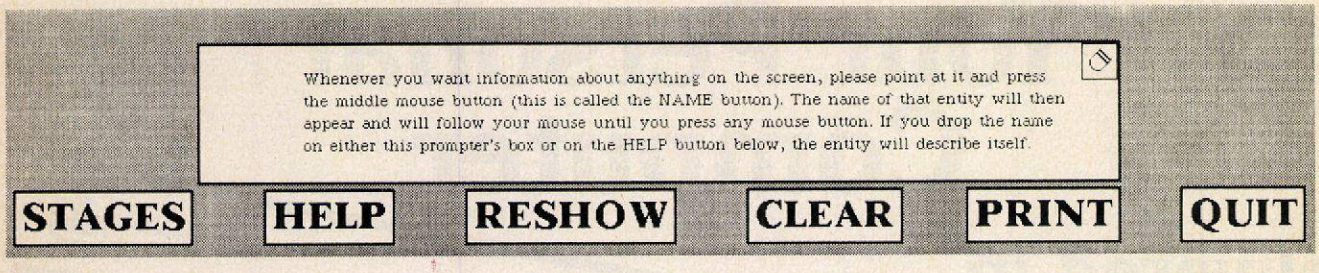


Figure 1: *The control panel and the prompter's box, showing an initial help message. The icon in the corner is an eraser.*

(*text continued from page* 188)

make some sort of word game. We'll follow their efforts, skimming over many of the details of their interactions with the Rehearsal World, with an eye to understanding some of the design decisions of Programming by Rehearsal itself. Although one person can manage both mouse and keyboard quite well, we'll assume that Laura is in charge of the mouse and Bill is typing on the keyboard. In what follows, the paragraphs describing the action of the designers have been italicized.

Bill and Laura know from their brief introduction to the Rehearsal World that all of the performers are clustered together in troupes waiting to be auditioned for parts in a production. They know also that the Rehearsal World includes a help facility that gives assistance and descriptive information about how to proceed.

Laura starts by selecting the HELP button from the control panel at the bottom of the screen (see figure 1). Selection of the HELP button causes the "prompter's box" to fill immediately with "procedural help" suggesting something that the designers might want to do next. When they select HELP initially, the procedural help message that appears explains that they can always obtain "descriptive help" about anything that they can see on the screen.

The fact that everything that can be seen is capable of self-description is an important component of the Rehearsal World and one that makes it accessible to nonprogrammers.

When they ask for descriptive help about the STAGES button, they learn that if they select the STAGES button, they will get a menu of troupes and productions. Laura selects the STAGES button which presents her with a menu of troupes and productions (see figure 2). She finds a Text performer in the Basic Troupe that she wants to audition to learn what it can do. Laura starts by asking it to describe itself and is told by the help system that if she selects the Text performer, she can edit the text that it displays. This editing is the default action of the Text performer. Laura and Bill spend a minute becoming familiar with the simple editor that the Text performer provides.

The Rehearsal World uses a three-button mouse for pointing at things on the screen. The SELECT mouse button causes a performer to execute its default action. The NAME button always causes the name of the entity to appear at the cursor point; if this name is dropped in the prompter's box, a description of the entity appears. Finally, the MENU button raises a pop-up menu for the performer, enabling the designer to send cues to it. In interacting with a finished production, only the SELECT button is used; that is, the NAME and MENU buttons are not needed by the student user.

Laura uses the MENU mouse button to see the category menu for the Text performer (see figure 3). Certain commonly used cues are at the top of this menu in lowercase, while others are grouped under categories in uppercase. Most of the cues and categories are shared by all performers. Only the

(*text continued on page* 192)

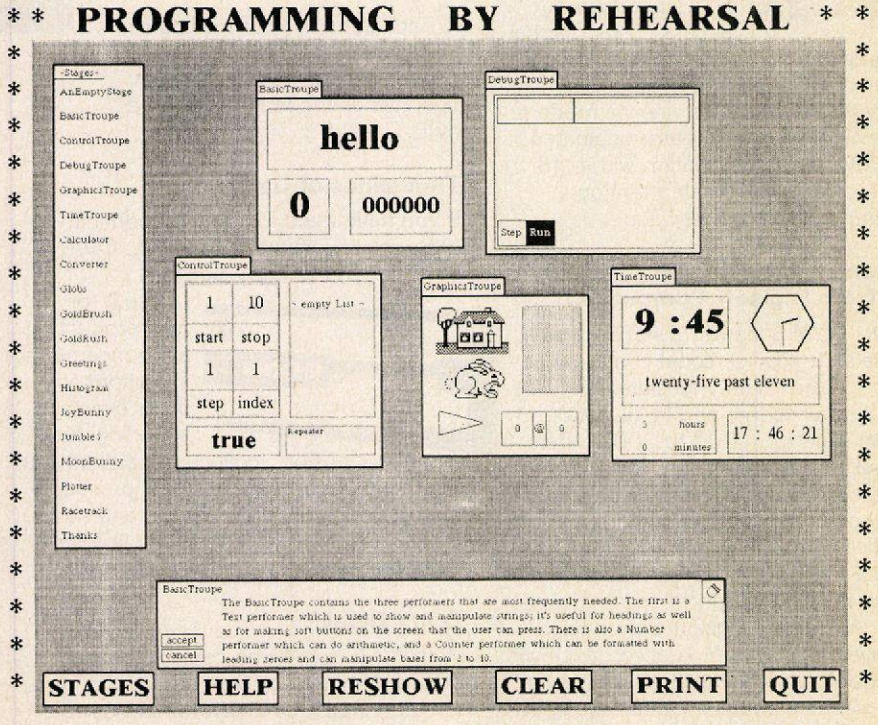


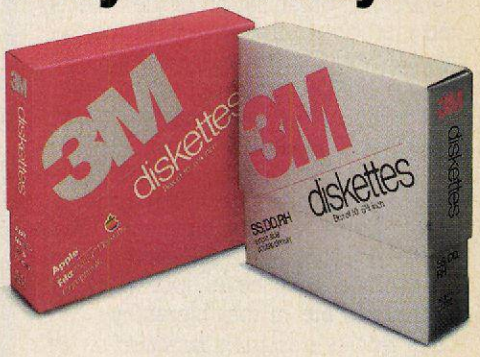
Figure 2: *The entire Rehearsal World theater, showing the* STAGES *menu at the left, all the available Troupes, and a descriptive help message about the BasicTroupe.*

You can count on 3M diskettes. Day after day.

Just like the sun, you can rely on 3M diskettes every day. At 3M, reliability is built into every diskette. We've been in the computer media business for over 30 years. And we've never settled in. We're constantly improving and perfecting our product line, from computer tape and data cartridges to floppy disks.

3M diskettes are made at 3M. That way, we have complete control over the entire manufacturing process. And you can have complete confidence in the reliability of every 3M diskette you buy.

Look in the Yellow Pages under Computer Supplies and Parts for the 3M distributor nearest you. In Canada, write 3M Canada, Inc., London, Ontario. If it's worth remembering, it's worth 3M diskettes.



Circle 331 on inquiry card.

3M hears you...

3M

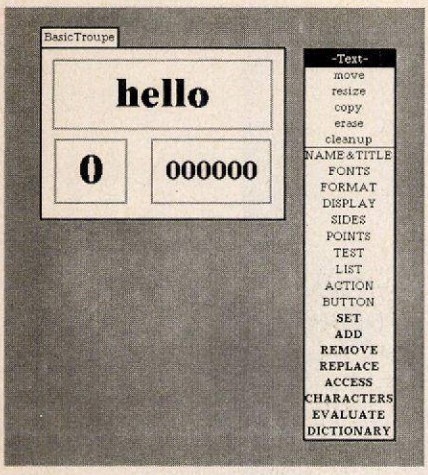


Figure 3: *A BasicTroupe, containing a Text, a Number and a Counter, and a category menu for the Text performer.*

(*text continued from page* 190)

categories at the bottom of the menu (in bold) are particular to the Text performer.

In its current prototype form, the Rehearsal World contains 18 primitive performers, each of which responds to a standard set of 53 cues and an average of 15 cues particular to that performer. To understand what this means, imagine a BASIC with a thousand reserved words. This complexity would be intolerable without a hierarchical organization and a simple way for the designer to browse that organization. The Smalltalk-80 system provides a window, called a Browser (see figure 4), whose visual structure reflects the hierarchical organization of the objects and methods in the system. In the Rehearsal World, functionality is organized around performers grouped together into troupes; the cues that each performer understands are grouped into categories. The result is that designers never have to scan too much information at a time, and, because each level in the hierarchy has a different screen appearance, they never lose track of where they are in that hierarchy.

Our novice designers proceed to rehearse the Text performer by sending it various cues. Laura tries move and resize and gets a pleasant surprise when the fonts change so that the text always fits within the performer's borders. She selects the SET category and gets a cue sheet showing the list of cues that have to do with setting text (see figure 5). Some cues, like setText:, take parameters that are indicated by parameter lines next to the cue. They use the help system to discover that they can type any string as a parameter to the setText: cue. Bill types 'goodbye' on the parameter line. When Laura selects the cue, "goodbye" appears in the Text performer.

They discover through rehearsal that the setJumbled cue produces a random permutation of the characters in the text. They enjoy looking at the different bizarre configurations that jumbling a word can produce and decide to explore no more, but to make a jumble game as their first design exercise. As often happens, interaction with the design environment itself leads to a creative idea.

One would not expect jumbling of text to be a basic capability of a programming language. A programmer who encountered a need for such a function would expect to write a simple routine. In a design environment, however, we expect to find a great deal of high-level functionality, chosen with care by the implementors of the environment, so that the designer's attention is not diverted from the design task itself.

Laura and Bill's initial idea for their simple production is to use two Text performers, one to be placed above the other on the stage. The top Text is to contain the word to be jumbled and the bottom one is to act as a soft button (a button on the screen which, when the student selects it with the mouse, causes something to occur). In this case its action will be to cause the jumbling of the top Text (see figure 6). Laura uses the copy cue to put a Text performer on an empty stage.

Any existing performer can be copied. Thus each performer acts as a prototype from which other performers can be generated; each new copy will have exactly the same characteristics as its prototype.

Laura and Bill use the resize cue to make the Text performer fill most of the top half of the stage, and then they copy it to make a second Text performer (exactly the same size as the first) in the bottom half of the stage. Bill types the word JUMBLE into it, as this is what they want the user to see. With the blocking thus completed, they decide to give each of their performers a mnemonic name that describes its purpose; they call the performers **JumbledWord** *and* **JumbleButton**. *Now they are ready to define the action of the bottom Text, which they want to act as a button.*

Any performer can become a button. By turning a performer into a button,

(*text continued on page* 194)

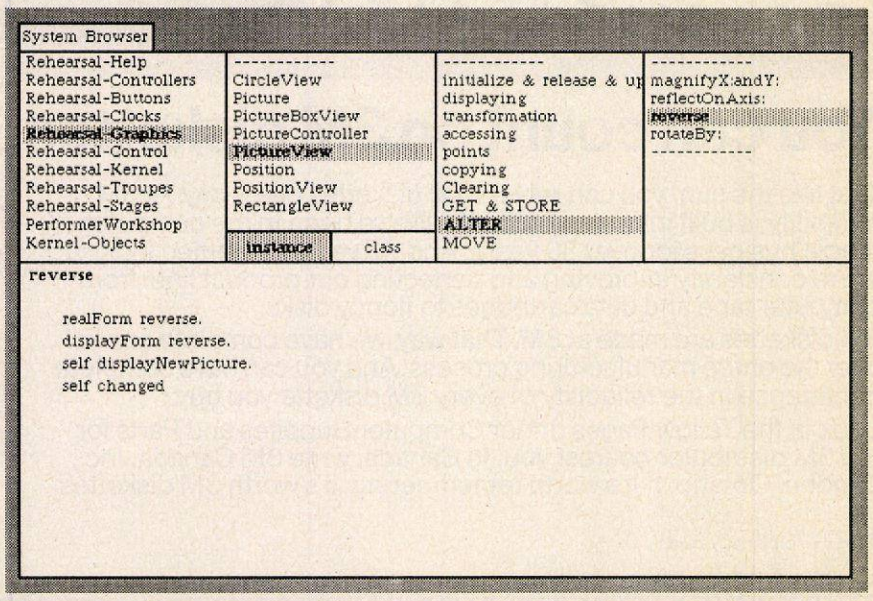


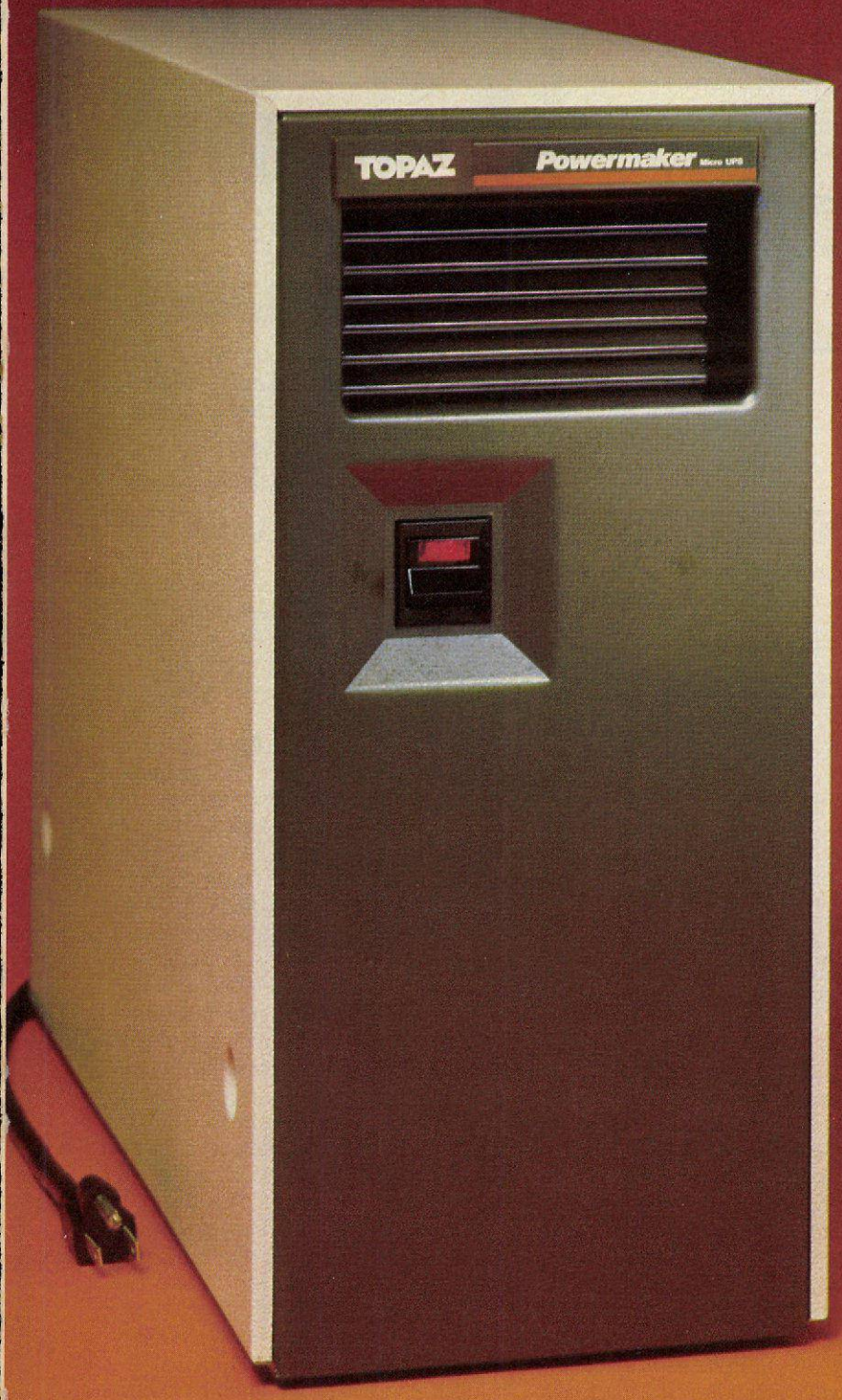
Figure 4: *A Smalltalk browser showing the* Rehearsal-Graphics *category, the* PictureView *class, its* ALTER *category, the message named* reverse *from that category, and the method associated with that message.*

Circle 335 on inquiry card. ⟶

Only from Topaz...

Powermaker® Micro UPS

Uninterruptible, computer-grade power —at half the cost



It's in a class by itself.

For about half the cost of other Uninterruptible Power Systems, you can now get the same degree of protection with our Powermaker Micro UPS. This remarkable new system eliminates computer problems caused by blackouts, brownouts, voltage sags and power-line noise.

Providing up to 75 minutes of continuous computer-grade power, our Powermaker Micro UPS is compatible with microcomputers and PC's. It's fully automatic, maintenance-free, portable and compact. It fits neatly alongside or under your desk or workstation. And because you can't always tell when you've lost primary power, our little UPS even features an audible line-loss alarm.

But best of all is the price. The Powermaker Micro UPS is priced right and is ready for immediate shipment. Find out more about our Powermaker Micro UPS. Call us at (619) 279-0831, or contact your local Square D distributor.

TOPAZ®

Excellence in Computer Power

SQUARE D COMPANY

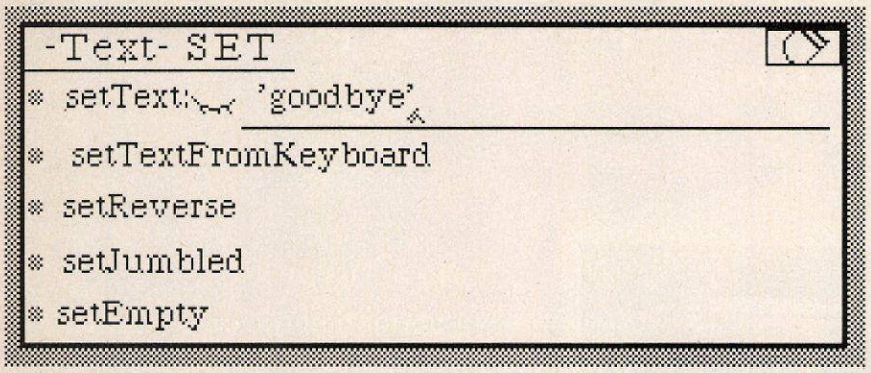


Figure 5: *A cue sheet for the* SET *category of a Text performer. The string 'goodbye' has been typed on the parameter line of its first cue.*

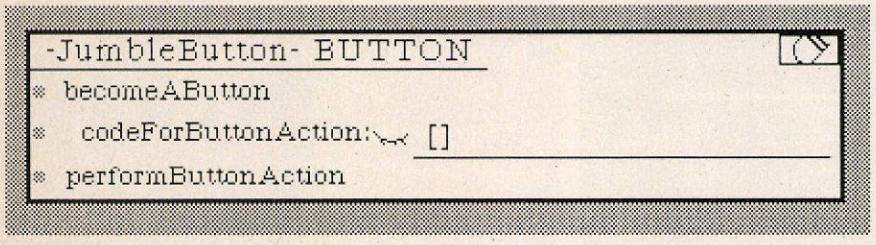


Figure 7: *The cue sheet for the* BUTTON *category of the performer named* JumbleButton. *The square brackets on the parameter line indicate that the designer should write some code between them.*

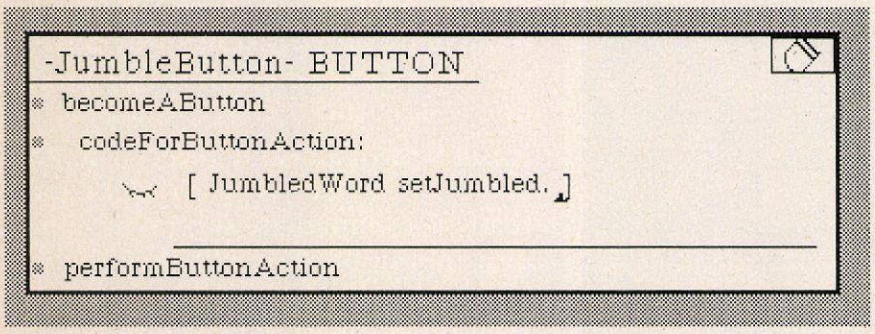


Figure 8: *The code, written by watching, which indicates what the* JumbleButton *should do whenever it is selected by the user.*

(*text continued from page* 192)

the designers get to decide what will happen when the user selects that performer. One of the categories on every category menu is BUTTON; its cue sheet contains the cue *becomeAButton (see figure 7).*

After Laura sends the becomeAButton is cue to the ***JumbleButton****, it no longer responds to selection by providing an editor; instead, it simply flashes. It is now a soft button on the screen, but it has no action. They must show it what to do.*

They do this by using the cue codeForButtonAction:|| to which every performer responds. Bill and Laura understand that they are expected to provide a block of code between the square brackets to describe the action that should occur when the user selects the ***JumbleButton****. The action they want is very simple; they just want the* ***JumbledWord*** *to receive the setJumbled cue. Bill knows that he does not have to type the code; instead the Rehearsal World will "watch" while they show it what to do.*

To the left of each parameter line is a tiny icon representing a closed eye. When Laura selects it, the eye opens to indicate that the system is indeed watching. Then Laura sends the setJumbled cue to the ***JumbledWord*** *by selecting it. The code* ***JumbledWord setJumbled*** *appears within the square brackets of the codeForButtonAction:|| cue of the* ***JumbleButton****, and the eye closes again (see figure 8).*

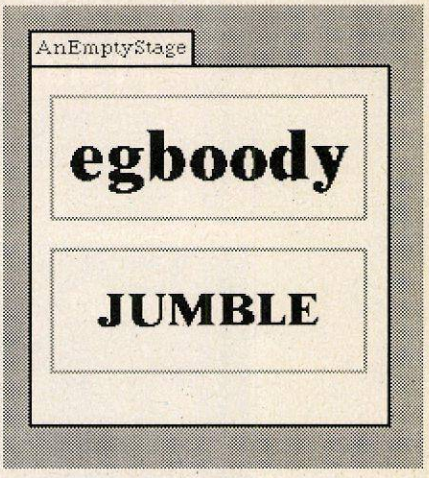


Figure 6: *A stage containing two Text performers, the top one showing a jumbled word and the bottom one acting as a button which the user can select to cause the jumbling to occur.*

Two significant obstacles to learning a programming language are mastering the language's syntax and learning the vocabulary. In the Rehearsal World, the designers rarely have to know either the syntax or the vocabulary as most writing of code is done by watching. While the eye is open, the designers rehearse a performer and the system makes a record of this rehearsal. The Rehearsal World's ability to watch, in combination with a mouse-driven interface, means that the designers do remarkably little typing. The designers know whether or not the code is correct not so much by reading it but by observing whether the effect produced on the stage is the desired one.

Immediately after Laura sends the codeForButtonAction:|| cue, she can select the newly defined button to see if it behaves as expected. Each time she selects the ***JumbleButton****, it flashes and the* ***JumbledWord*** *jumbles its text.*

In a traditional programming environment, the programmer moves back and forth between programming mode, in

(*text continued on page* 196)

COMPUTER WAREHOUSE

CALL TOLL FREE 1-800-528-1054

PRINTERS

Blue Chip
- M120/10 W/Commodore Interface ... $279
- M120/15 W/Commodore Interface ... $349

C-Itoh
- A10-20 ... $499
- F-10-Parallel or Serial ... $935
- 55 CPS Serial or Parallel ... $1319
- 8510 Parallel (Prowriter) ... $329
- 8510 SP ... $455
- 8510 SCP ... $525
- 8510 BPI ... $415

Computer International
- Daisywriter 2000 w/48K ... $985

Comrex
- CR-2 ... $449
- CR-2 Keyboard ... $150

Datasouth
- DS180 ... $1150
- DS220 ... $1499

Diablo
- 620 ... $815
- 630 API ... $1699
- 630 ECS/IBM ... $2075
- S-11 ... $559
- P-11 ... $559

Epson
- All Printer Models ... Call

Inforunner
- Riteman ... $249

IDS
- Microprism 480 ... $375
- Prism 132 ... $1310
- Prism 132 Color ... $1500

Juki
- 6100 ... Call

NEC
- PC-8023A ... $385
- PC-8025 ... $635
- 2010 ... $775
- 2015 ... $775
- 2050 ... $899
- 3510 ... $1365
- 3550 ... $1710
- 7710 ... $1900

Okidata
- 82A ... Call
- 83A ... Call
- 84P ... Call
- 84S ... Call
- 92 ... Call
- 93 ... Call
- 2350P ... Call
- 2410P ... Call

Panasonic
- 1090 ... Call
- 1091 ... Call
- 1092 ... Call

Qume
- 11/40 w/interface ... $1369
- 11/55 w/interface ... $1569
- Letter Pro 20P ... $609
- Letter Pro 20S ... $609

Silver Reed
- EXP400 ... Call
- EXP500P ... $385
- EXP500S ... $420
- EXP550P ... $480
- EXP550S ... $499

Star Micronics
- Gemini-10X ... Call
- Gemini-15X ... Call
- Delta 10 ... Call
- Delta 15 ... Call
- Radix ... Call

Tally
- MT 160L w/Tractors ... Call
- MT 180Lw/Tractors ... Call
- Spirit 80 ... Call

Toshiba
- P1350 Serial or Parallel ... $1429
- 1351 Serial or Parallel ... $1579
- 1340 ... $775

Transtar
- 120 Serial or Parallel ... $395
- 130 Serial or Parallel ... $549
- T315 ... $449

SANYO* EPSON SYSTEMS

DUAL DRIVE SYSTEM $1525
SANYO MBC-555 • SANYO CRT-36 HI-RES GREEN MONITOR EPSON RX-80 WordStar • CalcStar • Mailmerge • InfoStar • Spell Star • MS-DOS • Sanyo Basic

Above with Sanyo CRT-70 Color Monitor **$1939**

SINGLE DRIVE SYSTEM $1175
SANYO MBC-550 • SANYO CRT-36 HI-RES GREEN MONITOR • EPSON RX-80 WordStar • CalcStar • MS-DOS • Sanyo Basic

Above with Sanyo CRT-70 Color Monitor **$1629**

VIDEO TERMINALS

ADDS
- A-2 Green ... $490
- Viewpoint 60 ... $619

Altos
- Smart II ... Call

Hazeltine
- Esprit I ... $475
- Esprit II ... $485
- Esprit III ... $575

Qume
- QVT 102 Green ... $535
- QVT 102 Amber ... $550
- QVT 103 Green ... $840
- QVT 103 Amber ... $850
- QVT 108 Green ... $680
- QVT 108 Amber ... $699

Televideo
- 910+ ... $550
- 914 ... $515
- 924 ... $635
- 925 ... $700
- 950 ... $900
- 970 ... $985
- Personal Terminal ... $410

Wyse
- Wyse 50 ... $489
- Wyse 100 ... $680
- Wyse 300 ... $1020

Visual
- Visual 50 Green ... $619
- Visual 55 Green ... $709

Zenith
- Z-29 ... $644

QUADRAM
- Quadlink ... $449
- Quadboard 64K ... $265
- Quadboard 256K ... $450
- Quadboard II 64K ... $265
- Quadboard II 256K ... $450

MONITORS

Amdek
- Video 300 ... $130
- Video 300A ... $145
- 310A ... $160
- Color I Plus ... $275

Princeton Graphic
- HX-12 ... $499

Taxan
- 12" Amber ... $125

Zenith
- 12" Green Screen ... $95
- 12" Amber Screen ... $95

DISK DRIVES

Rana
- Elite 1 ... $215
- Elite 2 ... $345
- Elite 3 ... $410
- Controller (w/Drive only) ... $65
- 1000 w/DOS (for Atari) ... $305

DISKETTES

Maxell
- MD-1 (Qty. 100) ... $189
- MD-2 (Qty 100) ... $295

Scotch
- 744-0 (Qty. 100) ... $200

Elephant
- S/S S/D (Qty. 100) ... $155
- D/S D/D (Qty 100) ... $235

MODEMS

Hayes
- Smartmoden ... $199
- Smartmoden 1200 ... $485
- Smartmoden 1200B ... $430
- Micromodem IIe ... $235

US Robotics
- 212A Autodial ... $420
- Password 1200 ... $310
- IBM PC Modem ... $320

COMPUTERS

Altos
- All models ... Call

Columbia ... Call

Eagle
- PC-2 w/Monochrome Monitor ... $2699
- Spirit-2 ... $2525
- Spirit-XL ... $3675

NEC
- PC-8201A CPU ... $589
- PC-8206A 32K Ram ... $289
- PC-8281A Recorder ... $89
- PC-8201A-90 Battery Pack ... $15

Sanyo
- MBC-550 System ... $1175
- MBC-555 System ... $1525
- 1150 w/5000 printer ... $1575

Televideo Systems
- 802 H ... $4210
- 803 ... $1765
- 803H ... Call
- 1603 ... $2150
- 806/20 ... $4599
- 800 A (user station) ... $975
- TPC-1 ... $1525

Zenith
- Z-100 Low Profile ... $2625
- Z-100 All-In One ... $2800
- Z-150 Single Drive ... Call
- Z-150 Dual Drive ... Call
- Z-150 10 Megabyte ... Call
- Z-160 Single Drive ... Call
- Z-160 Dual Drive ... Call

COMMODORE
- 64 ... $227
- 1541 Disk Drive ... $239
- 1702 Monitor ... $239
- 1526 Printer ... $274
- 1530 Datasette (only w/64) ... $60

Order Line: 1-800-528-1054
Order Processing & Other Information: 602-954-6109

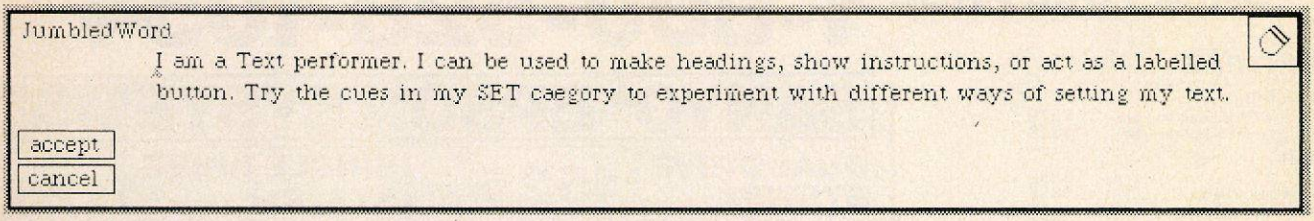
COMPUTER WAREHOUSE
2222 E. Indian School Rd.
Phoenix, Arizona 85016

AMERICAN EXPRESS

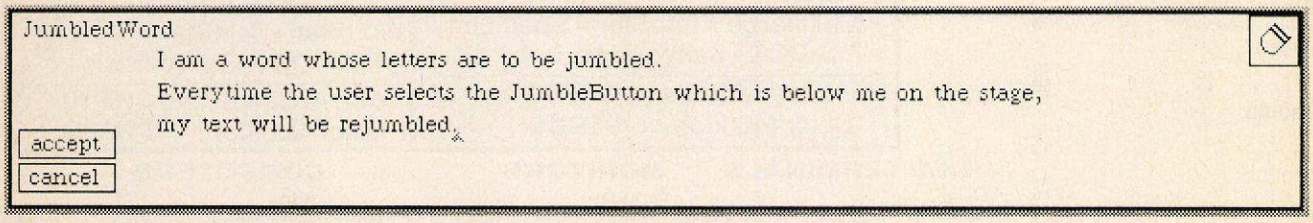
Store Hours: Mon-Fri 10-5:30 Saturday 9-1
Order Line Hours: Mon-Fri 8:30-5:30 Saturday 9-1

VISA MasterCard

Prices reflect 3% to 5% cash discount. Product shipped in factory cartons with manufacturer's warranty. Please add $8.00 per order for UPS shipping. Prices & availability subject to change without notice. Send cashier's check or money order...all other checks will delay shipping two weeks.



(9a)



(9b)

Figure 9: *The default comment associated with every Text performer (9a) and the edited comment to be associated only with the performer named* JumbledWord *(9b).*

(*text continued from page* 194)
which typing code is the dominant activity, and running mode, in which testing takes place. In Programming by Rehearsal, the designer does not feel any shift from one mode to another.

Even though their production is very simple, Laura and Bill decide to document it. They have already given the two Text performers appropriate names: ***JumbledWord*** *and* ***JumbleButttton****. They use the help system to get the default comment for the* ***JumbledWord*** *and edit it to be more specific (see figure 9).*

As a designer creates new productions and new performers, the Rehearsal World becomes more complex. The default descriptive help messages can be changed by the designer by simply editing what appears in the prompter's box and selecting the ACCEPT button. This provides a quick and pleasant method for providing descriptive comments for productions, performers, and cues.

It takes our two designers less time to produce their first jumble game than it takes to read about it. Although they have some ideas about how to make the game more interesting and educationally worthwhile, they decide to store what they have implemented so far. It is the stage itself that must be instructed to do the storing. The stage has its own category menu and one of its categories is STORE. They store their efforts under the name ***Jumble1*** *(see figure 10).*

No fixed set of functions provided in a design environment will ever be satisfactory; the designers will always run up against the limits of that set and wish for more capabilities. The fact that stages understand cues suggests one of the mechanisms for extensibility in the Rehearsal World: every stage can be

(*text continued on page* 198)

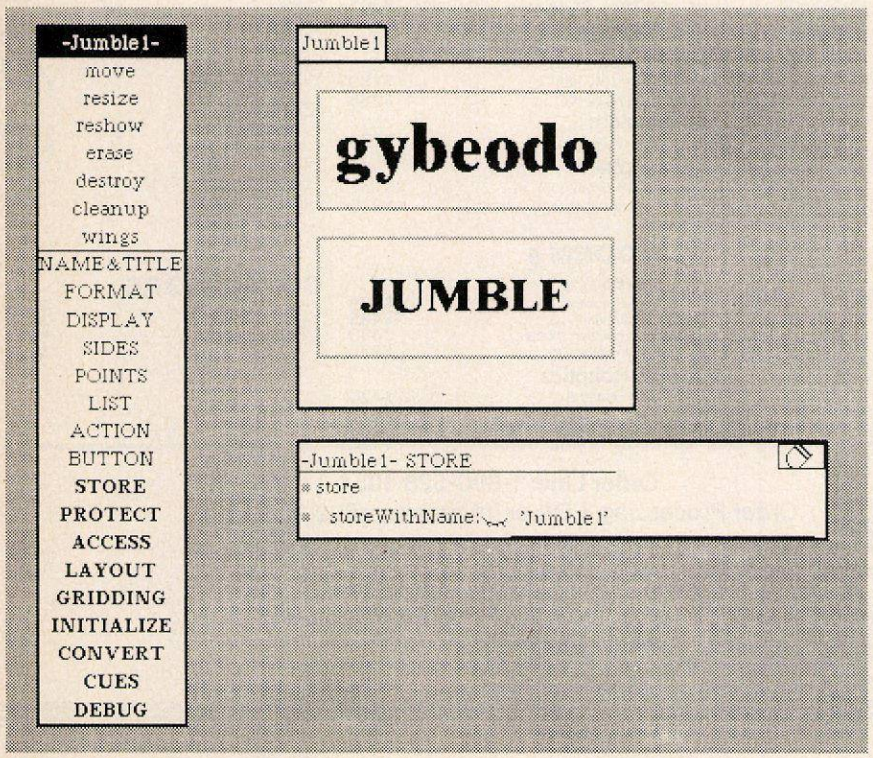
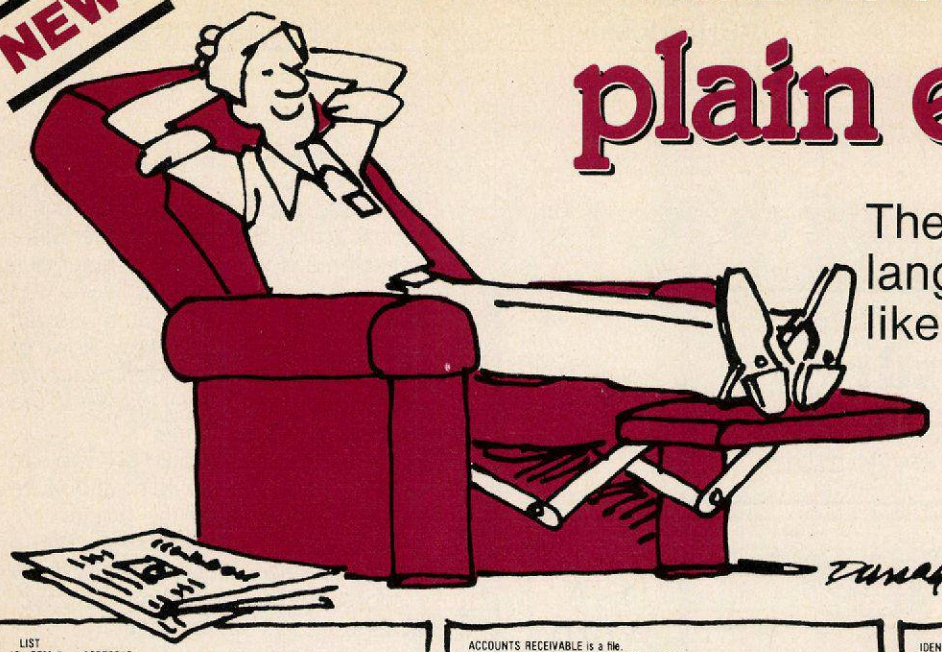


Figure 10: A *stage named* Jumblel; *it's a category menu and cue sheet for its* STORE *category.*

NEW!

FEEL AT HOME WITH plain english®



The first programming language that talks like you do!

```
 LIST
 10  REM #---ACREC010---
 20  REM #---CUSTOMER NAME AND ADDRESS LOAD PROGRAM---
 60  DIM B$(15)/DIM C$(20)/DIM D$(20)/A6$(1,8)="ZZ"
 70  J=0/OPEN #0, "CUSTMST,2"
 80  GOSUB 400
 90  J=6
100 GOSUB 400
110 N=0
120 !"LAST CUST # ENTERED WAS ",F,", ",A$,",",B$
130 !
140!"TO END PROGRAM ENTER 9999 AT CUST #"
150 J=J$1
160 !"ENTER FOLLOWING."
170 GOSUB 520
180 INPUT "CUT # ",F
190 IF F=9999 THEN 490
200 IF F =N THEN 220
210 N=F/GOTO 240
220 !"SEQUENCE ERROR-RETYPE"
230 GOTO 160
240 INPUT "1ST NAME ",A$(1,8)
250 IF A$=A6$ THEN 380
260 INPUT "LST NAME ",B$(1,15)
270 IF B$(1,2)=A6$(1,2) THEN 380
280 INPUT "ADRS LIN1 ",C$(1,20)
290 IF C$(1,2)=A6$(1,2) THEN 380
300 INPUT "ADRS LIN2 ",D$(1,20)
310 IF D$(1,2)^A6$(1,2) THEN 380
320 INPUT "TEL # ",E$(1,8)
330 IF E$(1,2)=A6$(1,2) THEN 380
340 INPUT "MO PMT AMT ",G
350 IF G=999 THEN 380
360 GOSUB 420
370 GOTO 150
380 N=N-1
390 GOTO 170
400 READ #0%96#J,A,F,A$,B$,C$,E$,G
410 RETURN
420 WRITE #0%96#J,A,F,A$,B$,C$,D$,E$,G
430 K=0/G=J
440 WRITE #0%96#J,A,F,A$,B$,C$,D$,E$,G NOENDMARK
450 RETURN
460 J=0
470 GOSUB 420
480 GOTO 140
490 REM #---CLOSE ROUTINE---
500 CLOSE #0
510 END
520 C$(1,20)="                    "/REM 20 BLANKS
530 D$=C$/A$=C$/B$=C$/E$=C$
540 RETURN
```

Basic

```
ACCOUNTS RECEIVABLE is a file.
    1  Uses        CUSTOMER NUMBER
    2  and         CUSTOMER NAME
    3  and         ADDRESS
    4  and         CITY STATE ZIP
    5  and         TELEPHONE NUMBER
    6  and         MONTHLY PAYMENT AMOUNT


ADD TO CUSTOMER FILE is a verb.
    1   Does       MESSAGE      "What is the customer number?"
    2   and        INPUT        CUSTOMER NUMBER
    3   and        MESSAGE      "What is the customer's name?"
    4   and        INPUT        CUSTOMER NAME
    5   and        MESSAGE      "What is the street address?"
    6   and        INPUT        ADDRESS
    7   and        MESSAGE      "What is the City State and Zip Code?"
    8   and        INPUT        CITY STATE ZIP
    9   and        MESSAGE      "What is the customer's phone number?"
   10   and        INPUT        TELEPHONE NUMBER
   11   and        MESSAGE      "What will the customer pay monthly?"
   12   and        INPUT        MONTHLY PAYMENT AMOUNT
   13   and        SAVE by CUSTOMER NUMBER in file ACCOUNTS RECEIVABLE
   14   and        REPEAT
```

plain english

```
IDENTIFICATION DIVISION.
PROGRAM—ID.
  TEST.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE—COMPUTER.    RMC.
OBJECT—COMPUTER.    RMC.
INPUT—OUTPUT SECTION.
FILE—CONTROL.
  SELECT AR—MASTER ASSIGN TO RANDOM "/u/files/armast"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS CUSTOMER—NUMBER.
DATA DIVISION.
FILE SECTION.
FD AR—MASTER LABEL RECORDS ARE STANDARD.
01  AR—REC.
    05 CUSTOMER—NUMBER                    PIC X(4).
    05 CUSTOMER—NAME                      PIC X(20).
    05 CUSTOMER—ADDRESS                   PIC X(40).
    05 CUSTOMER—CITY—STATE—ZIP            PIC X(40).
    05 CUSTOMER—PHONE                     PIC X(10).
    05 CUSTOMER—PAYMENT—AMOUNT            PIC 9(5).
WORKING—STORAGE SECTION.
PROCEDURE DIVISION.
RESIDENT SECTION 1.
STAR—UP.
  OPEN OUTPUT AR—MASTER.
LOOP.
  DISPLAY "ENTER CUSTOMER NUMBER OR  TO EXIT".
  ACCEPT CUSTOMER—NUMBER PROMPT.
  IF CUSTOMER—NUMBER = " " GO TO END—OF—JOB.
  DISPLAY "ENTER CUSTOMER NAME".
  ACCEPT CUSTOMER—NAME PROMPT.
  DISPLAY "ENTER CUSTOMER ADDRESS".
  ACCEPT CUSTOMER—ADDRESS PROMPT.
  DISPLAY "ENTER CUSTOMER CITY STATE ZIP".
  ACCEPT CUSTOMER—CITY—STATE—ZIP PROMPT.
  DISPLAY "ENTER TELEPHONE NUMBER".
  ACCEPT CUSTOMER—PHONE  PROMPT.
  DISPLAY "ENTER CUSTOMER PAYMENT AMOUNT".
  ACCEPT CUSTOMER—PAYMENT—AMOUNT  PROMPT.
  WRITE AR—REC INVALID KEY GO TO BAD—ADD.
  DISPLAY "CUSTOMER RECORD SAVED".
  GO TO LOOP.
BAD—ADD.
  DISPLAY "INVALID CUSTOMER".
  GO TO LOOP.
END—OF—JOB.
  CLOSE AR—MASTER.
  STOP RUN.
```

Cobol

Compare Plain English to any other language, as shown in the charts above. Straight forward plain english commands, using nouns and verbs are all that are necessary to create even the most sophisticated programs. Eliminate the complexities and rigid structures of the **old** traditional languages.

PERFECT FOR FIRST TIME PROGRAMMERS

Simple plain english statements are used to execute commands such as graphics, colors, sound and many more. An easy to understand TUTORIAL and helpful REFERENCE MANUAL (written in plain english by the way) will allow **anyone** to learn Plain English in as little as four hours.

FREE ACCOUNTING SOFTWARE.

For a limited time only, your Plain English package will include four accounting programs: Accounts Payable, General Ledger, Payroll and Mailing Lists. These programs may be customized **by you** to meet **your** requirements.

AVAILABLE TODAY AT YOUR LOCAL COMPUTER RETAILER

. . . PLAIN ENGLISH runs on all PC or MSDOS personal computers including Tandy's Model 2000 and requires only 192K memory and one 320KB floppy drive. You can also contact us directly for additional information, dealer inquiries invited.

AVAILABLE SOON . . . Special series for software developers including Unix versions.

plain english

a product of Common Language Systems Inc.

100 E. SYBELIA AVE. SUITE 375 MAITLAND, FL 32751 (305) 628-5973

Circle 64 on inquiry card.

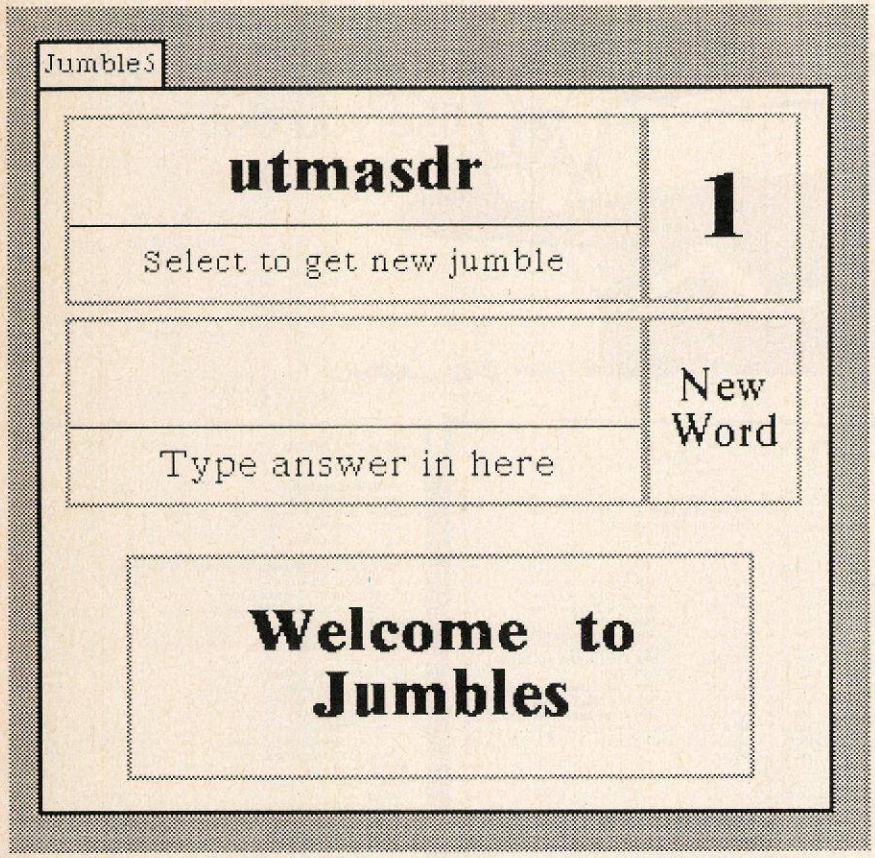


Figure 11: *An improved game named* Jumble5, *which evolved from* Jumble1.

(*text continued from page* 196)
converted into a new performer and every stage can be taught new cues. A designer who needs a new kind of performer can construct one by aggregating existing performers on a stage, teaching that stage some appropriate new cues, and converting the result into a new performer.

There are many circumstances in which the designers may wish to aggregate performers: several performers belong together as a logical and spatial unit; a group of performers are to be used repeatedly within a production or in several different productions; a production is very complex, and creating a new performer allows a factorization of the entire problem into smaller ones.

Bill and Laura's jumble game goes through four revisions until it finally becomes the one shown in figure 11. This improved game contains four Text performers and a Number performer. The large Text at the bottom is used simply to give feedback to the student. The Text labeled "New Word" has been turned into a button; its button action is to cause a new secret word to be chosen from a List and presented in jumbled form in the top Text performer. This performer has also been turned into a button; its button action is to rejumble itself. The number of rejumblings is shown by the Number performer next to it. The Text performer in the center of the stage is to be edited by the student who will type the answer there. Every time that Text is changed, it will cause the answer to be checked against the secret word and suitable feedback to be provided. It does this by means of its change action.

When a performer changes in some fundamental way, as when a Number performer changes its value or a Text performer changes its text, it executes its change action. The default change action of a performer is to do nothing, but the designer can define this action for any performer. Certain other performers have additional possible actions: the Repeater performer has a repeat action, the List performer has a selection action, and the Traveler performer has a move action.

In the Jumble5 game, Laura and Bill use a List performer to keep a list of secret words. Since they don't want the user to see the List, they place it in the wings (see figure 12).

While everything should be visible to the designers, not everything should be visible to the user of the production. Wings can hold performers waiting to appear on stage, data structures like the List of secret words, or temporary variables used in computations.

A very simple game grew and prospered as our designers implemented it, changing in response to their new understanding of what they were doing, and to the needs and interests of users and other designers who experimented with it. It became something real that people wish to play with and from which they can get some increased intuitive understanding of the rules underlying English orthography.

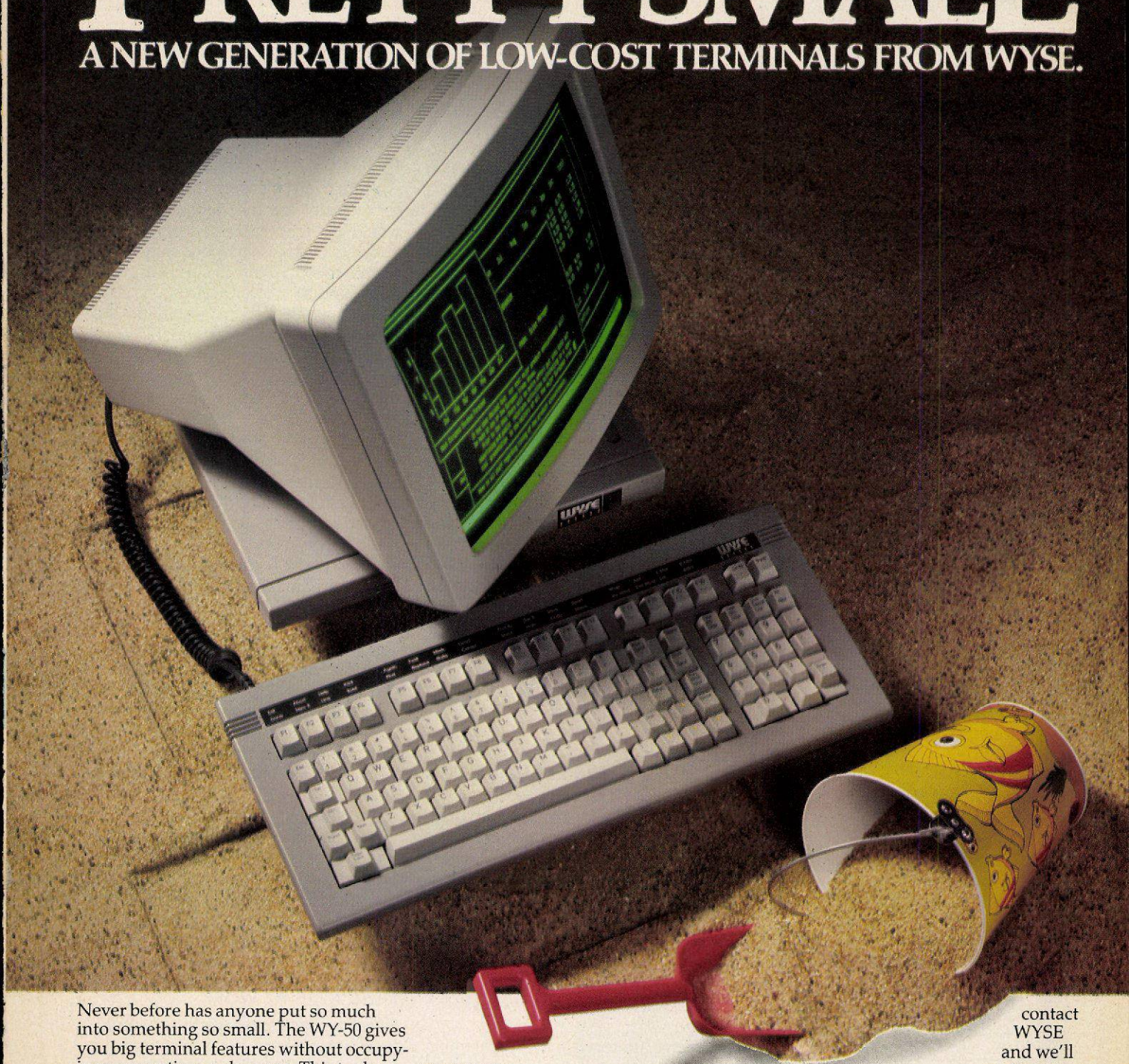
Beneath the Rehearsal World — Through the Trapdoor

The Rehearsal World in some ways may be thought of as a visible Smalltalk. Although our original intention was to remove the need for programming at the Smalltalk level, it is paradoxically true that the Rehearsal World provides an excellent entry point for an incipient Smalltalk programmer. Designers may drop through the trapdoor of the Rehearsal World; beneath they will find all the tools of the Smalltalk-80 programming environment. A Rehearsal World tool found there is called the Performer Workshop. It looks like a simplified Smalltalk browser and provides a mid-level mechanism for creating new primitive performers and defining new cues.

For each kind of performer there is a corresponding Smalltalk class that is a subclass of class **Performer**. The inheritance mechanism of Smalltalk allows the subclass to inherit the message interface of class **Performer**. Each production corresponds to a subclass of class **Stage**. When designers store a production, the Rehearsal World defines a new subclass of class **Stage**. Interest-

(*text continued on page* 200)

THE WY-50.
PRETTY SMALL
A NEW GENERATION OF LOW-COST TERMINALS FROM WYSE.



Never before has anyone put so much into something so small. The WY-50 gives you big terminal features without occupying your entire work-space. This took revolutionary design. Design a lot of people couldn't accomplish for the price. But we did.

In fact, the WY-50 introduces a new standard for low-cost terminals. You get a compact, full-featured design that meets the most advanced European ergonomic standards. 30% more viewing area than standard screens. And a price tag as small as they come.

The WY-50 sells for only $695.00.

FEATURES:

- **14" screen.**
- **80/132 column format.**
- **Soft-set up mode.**
- **High resolution characters.**
- **Low-profile keyboard.**
- **Industry compatible.**
- **Only $695.00.**

For more information on the revolutionary design, outstanding features and unique good looks of the new WY-50, contact WYSE and we'll send you a brochure filled with everything you need to know. The WY-50. The full-featured terminal with the small price.

WYSE Circle 359 on inquiry card.

Make the Wyse Decision.

WYSE TECHNOLOGY *3040 N. First St., San Jose, CA 95134, 408/946-3075, TLX 910-338-2251, Outside CA call toll-free, 800/421-1058, in So. CA 213/340-2013.*

(text continued from page 198)
ingly, a stage is so much like a performer that class **Stage** is actually a subclass of class **Performer**.

When designers create new performers, the Rehearsal World defines a new subclass of **Performer** and writes the code for the appropriate additional methods that the class will need for layout and for cues. Because the code written by the Rehearsal World is indistinguishable from code written by a programmer, one can inspect it and modify it in either a Performer Workshop or a Smalltalk browser (see figure 4).

There are two important features of Smalltalk that are not present in the Rehearsal World. The first is the ability to create a hierarchy of objects. In Smalltalk, when one constructs a new kind of object—that is, a class—one usually constructs it by defining a subclass of the existing class that is most like the new class. In that way the new class can inherit a great deal of the desired behavior. In the Rehearsal World, there is no concept of class. A designer who wants a new production that is similar to an existing one can modify the existing production and store it under a different name. A major weakness of this method is that modifications made to the first production will not be automatically reflected in the modified one. In contrast, a modification made to a Smalltalk class will be automatically reflected in its subclasses.

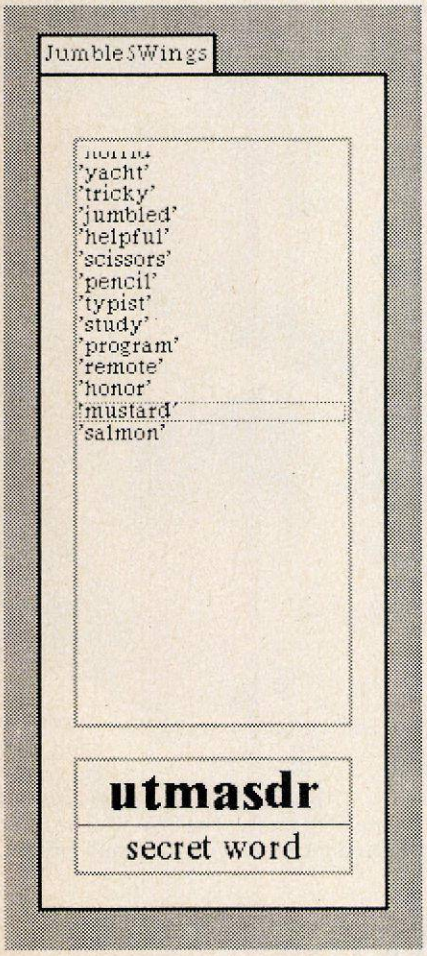


Figure 12: *The wings of the* **Jumble5** *game, showing a List performer in which the current secret word is selected.*

The second difference between Smalltalk and the Rehearsal World is that in Smalltalk there is a distinction between a class and an instance of that class. The class is the abstraction; an object is always an instance of some class. A class may have any number of instances. Any changes to the class will be immediately reflected in all its instances. In the Rehearsal World, there are no abstractions, thus no classes. Everything is visible. Any performer can serve as a prototype and one gets new performers through copying. What is lost is the ability to have changes made to the original reflected automatically in the copies.

Debugging

Ordinarily, the sooner a program gives evidence that something is wrong, the easier it is for the programmer to diagnose the problem. Designers in the Rehearsal World find that bugs manifest themselves very quickly because nearly all state information is visible and because the flow of control from performer to performer is fairly obvious to the eye. Even so, a situation will occasionally arise in which the designer cannot easily account for some behavior on a stage.

It seems appropriate in Programming by Rehearsal that help should come in the form of another performer, the Debugger performer (see figure 13). A Debugger, when placed on a stage, intercepts all the actions that performers execute, shows their code, and waits for the designer to tell it to go on. While the actions of the production are thus halted, the designers can investigate the cause of a problem using any of the normal Rehearsal World activities such as opening up cue sheets and sending cues. Additional actions that may be initiated are placed in the Debugger's queue for later execution.

Animation and Multiple Processes

An intuitively pleasing, though incorrect, model for the Rehearsal World would be that each performer goes about its business independently of the others except when it needs another performer to answer a question or do something. Performers would be like people in the real world, capable of independent action but interacting through requests. Animation, you might think, would be easy because each performer would have its own rules for moving around on the screen. In this model, which we call the one-process-per-performer model, each performer would essentially have its own processor for its private use. Trouble comes when performers have to share resources and coordinate that sharing. Several schemes for dealing with these problems have been developed over the years.

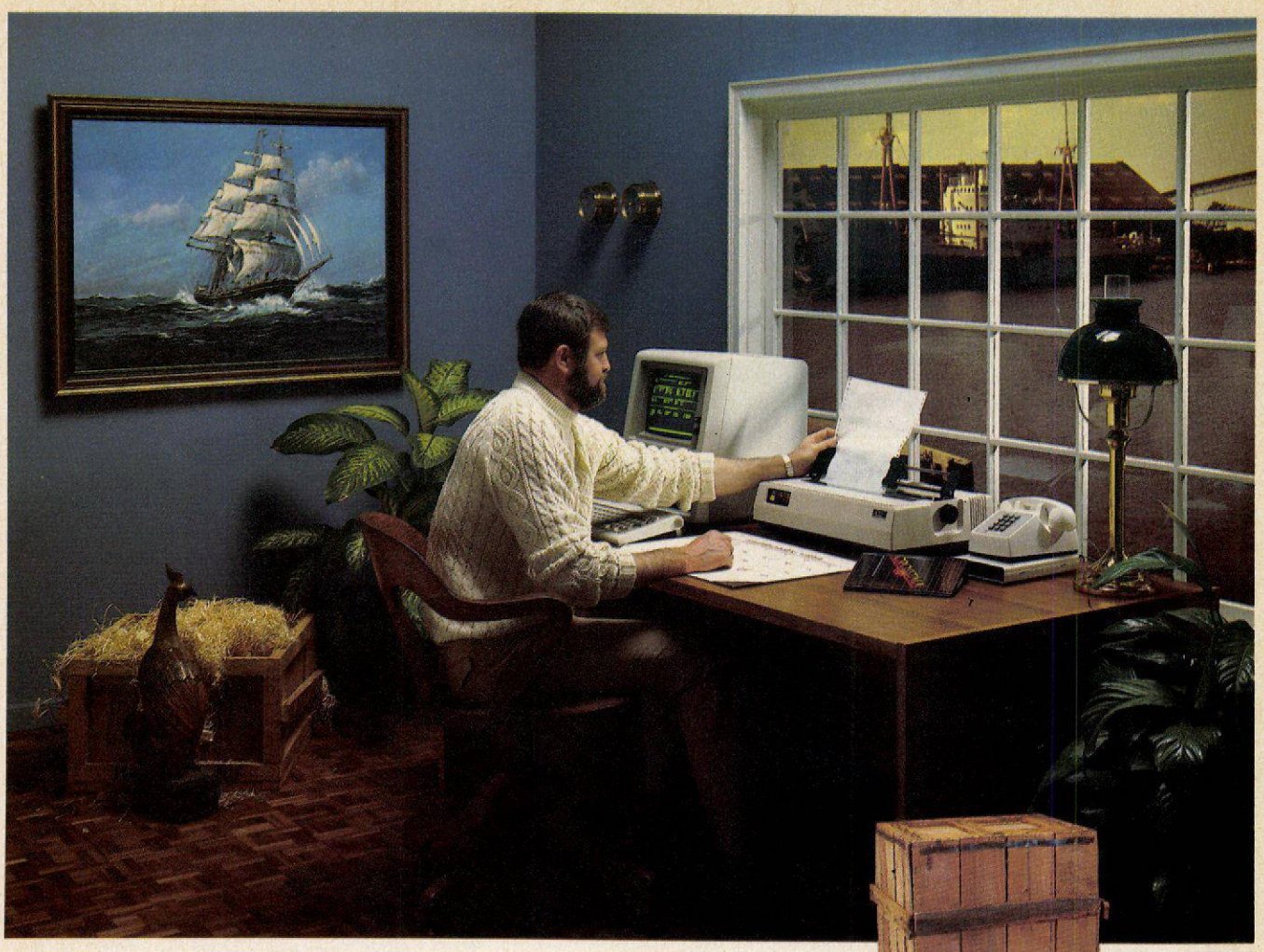
Our own solution to the problems introduced by having one process per performer was to allow each user action to initiate a single independent process that either runs to completion or, as with animation, continues in an infinite loop. A single production can, at any given time, have any number of different processes running in it. (Beyond that, there can be several stages on the screen at a time, each running its own processes.) This one-process-per-user-action model has so far proven to be both intuitive and powerful, though we see it as an area where further research is necessary.

Designers at Work

Since the Rehearsal World is a prototype system, very few designers have had a chance to experiment with it. The first one to actually use the system was Joan Ross, a curriculum designer from the University of Michigan. Joan created many interesting productions using the Picture and Turtle performers. She helped us to debug the system and to understand how to improve it on all levels as we prepared for a pilot study.

We spent a month responding to the

(text continued on page 202)



International Connections

With the industry's most popular data communications program, the world is at your command.

An import/export office in New Jersey can instantly check the London market for current dollar exchange rates ... send Hong Kong an updated production schedule ... print-out the week's sales results from the Dallas branch.

There's virtually no limit to how far you can reach with your microcomputer, ordinary telephone lines, and CROSSTALK.

Even if your own business and personal needs are closer to home, you'll appreciate CROSSTALK's compatibility with a wide user base ... smart terminal characteristics ... total modem control ... and the ability to capture data at a high speed for later off-line editing. CROSSTALK has extras you may not find in other programs. Data capture to memory buffer (and on-line display). Protocol error-checking file transfer. Modem/telephone hangup, and display of elapsed time of call. Command file power and flexibility. Remote takeover and operation. And much more.

There is a CROSSTALK version for almost every CP/M, CP/M-86, or IBM DOS based microcomputer system. See your dealer, or write for a brochure.

CROSSTALK™

MICROSTUF®

1845 The Exchange / Atlanta, Georgia 30339 / (404) 952-0267

CROSSTALK is a trademark of Microstuf, Inc., CP/M and CP/M-86 are trademarks of Digital Research, Inc., IBM is a trademark of International Business Machines, Inc.

Circle 224 on inquiry card.

(text continued from page 200)
issues that Joan raised as a result of her experiences and then invited Dan Fendel and Diane Resek, curriculum designers and faculty members of the Mathematics Department at San Francisco State University, to visit for three days to see what they could create in the Rehearsal World. They are very experienced designers, familiar with the power of interactive computer graphics, but they are not programmers.

We gave them a tour of the system and within 45 minutes Dan and Diane had taken over and were using the Rehearsal World themselves. They started by investigating a simple production we had made about probability and soon suggested and implemented some improvements. They found out how it worked by looking at the button actions and change actions of the performers, both on stage and in the wings. By the end of the first afternoon, they had turned it into a game that bore only a slight resemblance to our original exploratory activity. In the process, they had auditioned Texts, Numbers, Lists, and Repeaters to discover their capabilities, dealt some with the blocking of the stage, written a fair amount of code by watching, and understood about button actions, change actions, and repeat actions.

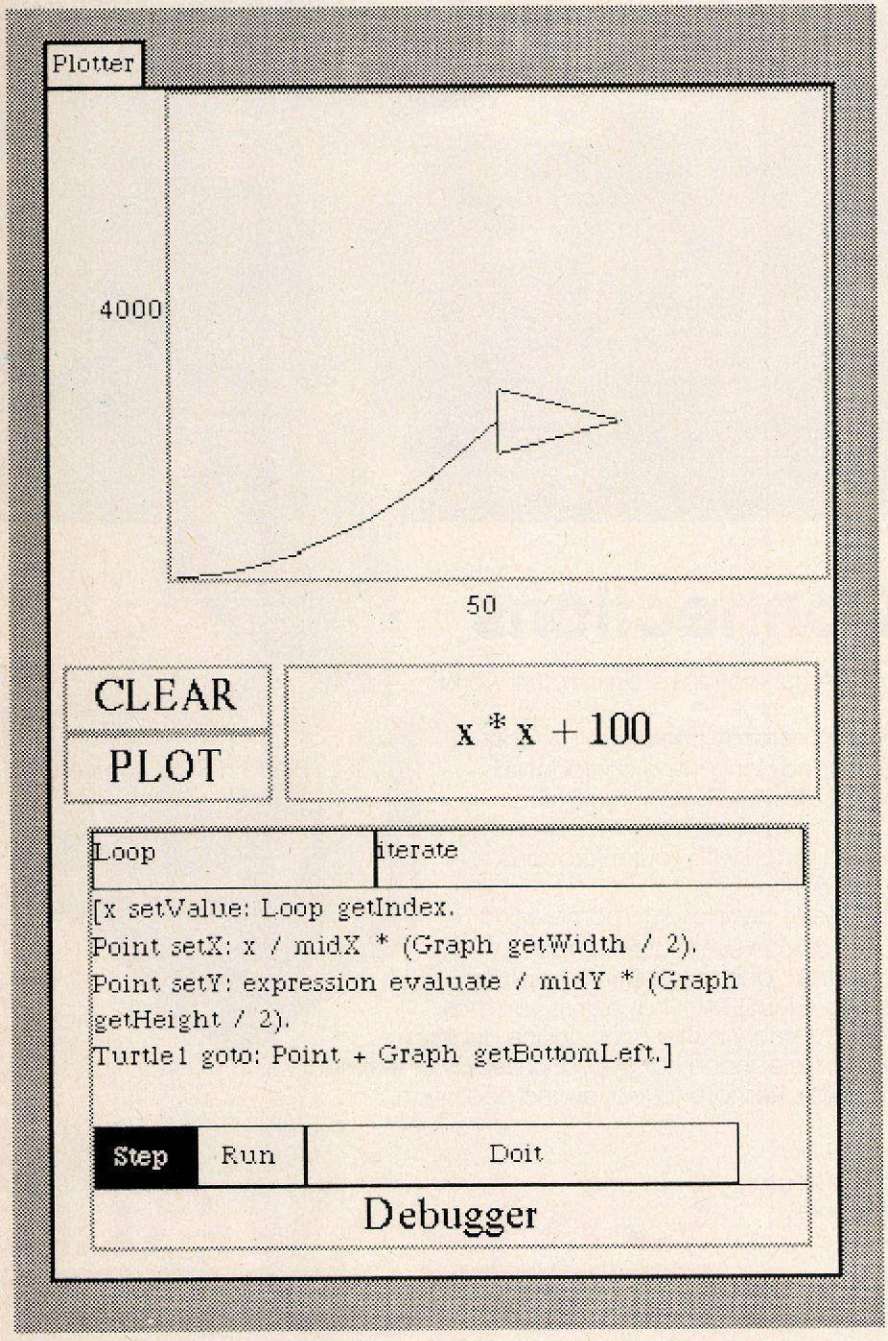


Figure 13: A *stage on which a Debugger performer has been placed temporarily so that the designer may observe the code for each successive action.*

Dan and Diane spent an hour the next morning away from the machine, designing with words and a pencil. In the course of this design session, they refined their embryonic ideas for a fraction game through discussion of both the pedagogical issues and the fantasy through which they should be transmitted. They also considered which Rehearsal World performers they would need in their proposed game. The fantasy involved a cave filled with gold dust. They envisioned the ceiling of the cave as an irregular set of stalactites; they saw the floor as tiled. The student's problem would be to sweep a vertical broom through this cave, one floor tile at a time, trying to collect as much gold dust as possible without ever allowing the broom to touch the ceiling. The broom would stretch or shrink by a certain fractional amount which the student would specify before each move. For example, if the student edited the fraction to read 2/1, the broom would become twice as tall when it moved.

They had other design criteria as well. They wanted the game to configure itself differently every time the START button was selected, and they also wanted to make it easy for a designer to specify an easy cave, with broad floor tiles and very little variation in the ceiling, or a hard one. They wanted to have a score that was expressed as a percentage of the available gold dust; they wanted some sort of disaster to occur if the student made the fraction too large and the broom touched the ceiling. They decided to call their production GoldRush (see figure 14).

We found this description quite overwhelming for an initial project, as we

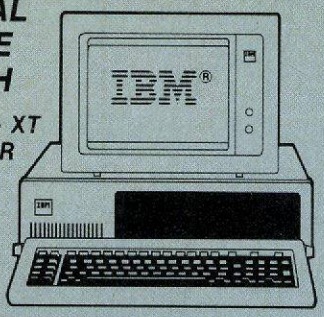
(text continued on page 204)

COMPUTER HUT™

COMPARE OUR SERVICE & PRICE!

SPECIAL OF THE MONTH

IBM-PC & XT CALL FOR PRICE



Tandon TM100-2 DS/DD $225
PANASONIC JA 551
SHUGART SA-455 half-high BEST
TEAC FD-55B Slimline PRICES
AMDEK half-high
HITACHI half-high

MAYNARD ELECTRONICS
Floppy Disk Controller $169
FDC w/Par. Port $219
FDC w/Ser Port $239
SANDSTAR SERIES CALL

QUADRAM
Quadboard-PP,SP,C/C,Mem + s/w
Expandable to 384K . . . . . CALL
Quad 512 + SP,Mem with s/w
64K $249
Quadcolor . . . . . CALL

AST RESEARCH
MegaPlus II 4-Funct 64K + s/w $279
6-Pack 5-Funct 64K + s/w $279
I/O Plus $129

TECMAR
Graphics 720 × 400 16 colors $529

HERCULES
Hi Res Graphics 720 × 348 $359

FREDRICKS ELECTRONICS
COLORPLUS 640 × 200,
16-Color + s/w $399

MA SYSTEMS
PC Peacock w/Par Port $275

MICROLOG
Baby Blue $359
Baby Blue II 64K $575

PARADISE
Multidisplay $395

HARD DISK - IBM-PC & XT

MOUNTAIN — External Syst.
5MB . . . . $1539 10MB $1799
15MB . . . . $2309 20MB $2549
20M Tape back up $1695
MAYNARD . . . . . CALL

PRINTERS

EPSON
FX80 . . . . . CALL FX100 . . . CALL

brother
HR1 A Par $599
HR-15 Par $459
HR-25 . . . . . CALL

DYNAX
DX-15 Par . . . . $459 Ser $489

C-ITOH
STARWRITER A-10 CALL
STARWRITER F-10 P $1095
PROWRITER 8510 SP 180 CPS $649

star MICRONICS
Gemini 10X . . . $299 15X . . . $399

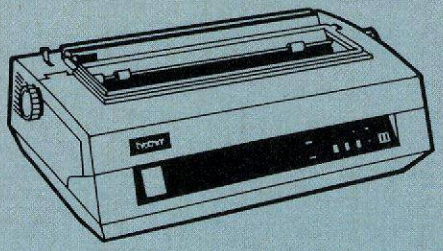
OKIDATA

82A	CALL	83A	CALL
84P	BEST	84S	BEST
92P	PRICES	92S	PRICES
93P		93S	

NEC

3510	$1485	7710	$1995
3515	$1479	7715	$2039
3530	$1575	7720	$2495
3550	$1695	7730	$1995
2000 Series			CALL

TOSHIBA
P1351 . . . . . $1649
P1340 . . . . . $849
IDS, DAISYWRITER . . . . . CALL



MODEMS

HAYES
Smartmodem 1200 $489
Smartmodem 1200B $419
NOVATION, US ROBOTICS CALL

COMPUTERS

eagle CALL
COLUMBIA DATA PRODUCTS, INC. CALL
CORONA CALL
TAVA PC CALL
COMPAQ CALL

MONITORS

AMDEK
Video 300G . . . $145 300A $155
Video 310A $189
Color II . . . . . $429 Color II + CALL
PGS
HX12 Hi Res RGB monitor . . . . BEST PRICES
MAX-12 Hi Res Mono.
SR-12 Super Hi Res RGB

SOFTWARE FOR IBM-PC

Word Perfect $299 WordStar $275
DBase II . . . . $389 VisiCalc $189
Multiplan . . . $175 Multimate . . . $299
MICROSOFT Word $269
AND LOTS MORE

CANADIAN COMPUTER HUT

AUTHORIZED DEALER
MICROCONTEXT INC.
5253 AVE DU PARC
MONTREAL
QUE H2V4P2.
(514) 279-7291

Published Prices are for U.S.A. Only
Please call for Canadian Prices

ANY PRODUCT NOT LISTED? CALL

COMPUTER HUT
OF NEW ENGLAND INC.
101 Elm St., Nashua, NH 03060

ORDERS & INFORMATION
(603)889-0666

ORDER-LINE ONLY PLEASE
(800) 525-5012

All products usually in stock for immediate shipment and carry full manufacturers' warranty. Price subject to change — this ad prepared two months in advance. You get the lowest price. We honor personal checks — allow 10 days to clear. COD up to $300 add 3%. Visa, MasterCard add 3%. For shipping & insurance add 3% or $5.00 min. for small items and $10 min for monitors, printers, etc. APO & FPO orders add 12%. Include phone number.

IBM is a trademark of IBM Corp.

Return authorization and order status (603) 889-7625

Circle 76 on inquiry card.

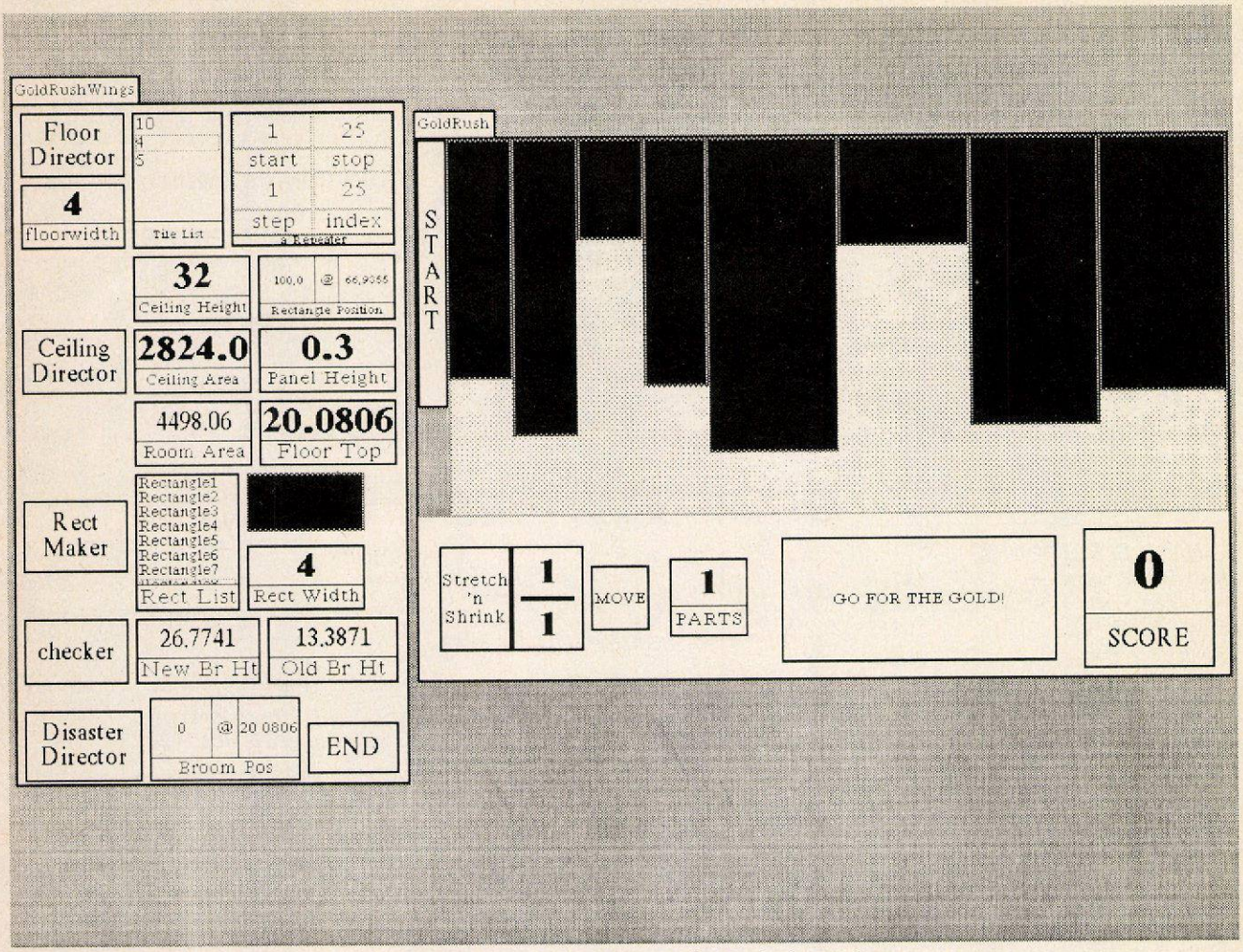


Figure 14: *The GoldRush game and its complicated wings, showing more performers backstage than are on stage.*

(*text continued from page* 202)

had expected them to embark on something at the level of the Jumble Game described earlier. Rather than starting with a toy example for practice, they were embarking on a real-world task after only one day's experience. We worried that they had chosen something too difficult for them to accomplish in the remaining two days.

By lunch time they had figured out how to use the Turtle to draw the floor. They said, "We need a Floor Director to be in charge of drawing the floor," and placed a button in the wings labeled FloorDirector for that purpose. They used this same strategy to make a CeilingDirector, a Checker to test whether or not the broom was touching the ceiling, and a DisasterDirector in charge of what should happen when it did. Certain performers had become, if you will, visible procedures. They invented this strategy on their own, led to it by the Rehearsal World's emphasis on buttons.

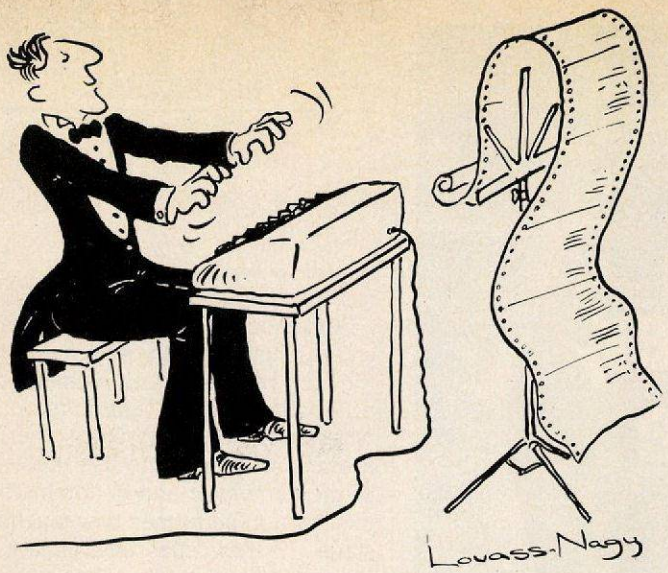
Next to these directors in the wings, they placed the performers that would be needed by the directors to accomplish their tasks. These performers fulfil the role of variables; since everything in the Rehearsal World must be visible, all variables must be represented by performers. By grouping their performers in a logical manner, they could debug their program easily by selecting a button, like the CeilingDirector, and simply watching what happened, both on stage and in the wings.

Their next task was to implement the broom (for which they used a Rectangle), the START button, and the MOVE button. The action of the START button was simply to cause the FloorDirector and the CeilingDirector to perform their button actions. The action of the MOVE button was first to move the broom and then to ask the Checker to determine whether or not the broom was touching the ceiling. If it was, it asked the DisasterDirector to perform its action; if it wasn't, the Checker computed the score. That they had not yet even designed the disaster didn't matter; they were using top-down programming techniques, realizing that they could return later and replace the empty code block of the DisasterDirector with whatever they wanted.

By the end of the day, the FloorDirector and the CeilingDirector were both working properly and they could move the broom through the cave. They started to plan the randomness that they wanted to build into the button action of the START button.

The next day they made a fraction to be edited by the user, creating it from two Numbers and two Rectangles, one to act as the line between the Numbers, the other to act as a frame. This looked and worked fine, but they soon discovered that it was a great disadvantage to be dealing with four independent performers instead of a single unified one: whenever they decided that their fraction was the wrong size or in the wrong place, they had to resize or move

(*text continued on page* 206)



Before Johann Sebastian Bach developed a new method of tuning, you had to change instruments practically every time you wanted to change keys. Very difficult.

Before Avocet introduced its family of cross-assemblers, developing micro-processor software was much the same. You needed a separate development system for practically every type of processor. Very difficult and very expensive.

But with Avocet's cross-assemblers, a single computer can develop software for virtually any microprocessor! Does that put us in a league with Bach? You decide.

The Well-Tempered Cross-Assembler

Development Tools That Work

Avocet cross-assemblers are fast, reliable and user-proven in over 3 years of actual use. Ask NASA, IBM, XEROX or the hundreds of other organizations that use them. Every time you see a new microprocessor-based product, there's a good chance it was developed with Avocet cross-assemblers.

Avocet cross-assemblers are easy to use. They run on any computer with CP/M* and process assembly language for the most popular microprocessor families.

5¼" disk formats available at no extra cost include Osborne, Xerox, H-P, IBM PC, Kaypro, North Star, Zenith, Televideo, Otrona, DEC.

Turn Your Computer Into A Complete Development System

Of course, there's more. Avocet has the tools you need from start to finish to enter, assemble and test your software and finally cast it in EPROM:

Text Editor VEDIT -- full-screen text editor by CompuView. Makes source code entry a snap. Full-screen text editing, plus TECO-like macro facility for repetitive tasks. Pre-configured for over 40 terminals and personal computers as well as in user-configurable form.

CP/M-80 version $150
CP/M-86 or MDOS version $195
(when ordered with any Avocet product)

EPROM Programmer -- Model 7128 EPROM Programmer by GTek programs most EPROMS without the need for personality modules. Self-contained power supply ... accepts ASCII commands and data from any computer through RS 232 serial interface. Cross-assembler hex object files can be down-loaded directly. Commands include verify and read, as well as partial programming.

PROM types supported: 2508, 2758, 2516, 2716, 2532, 2732, 2732A, 27C32, MCM8766, 2564, 2764, 27C64, 27128, 8748, 8741, 8749, 8742, 8751, 8755, plus Seeq and Xicor EEPROMS.

Avocet Cross-assembler	Target Microprocessor	CP/M-80 Version	CP/M-86 IBM PC, MSDOS** Versions
XASMZ80	Z-80	$200.00 each	$250.00 each
XASM85	8085		
XASM05	6805		
XASM09	6809		
XASM18	1802		
XASM48	8048/8041		
XASM51	8051		
XASM65	6502		
XASM68	6800/01		
XASMZ8	Z8		
XASMF8	F8/3870		$300.00 each
XASM400	COP400		
XASM75	NEC 7500	$500.00	
Coming soon: XASM68K...68000			

(Upgrade kits will be available for new PROM types as they are introduced.)

Programmer $429
Options include:
Software Driver Package -- enhanced features, no installation required.
CP/M-80 Version ..... $ 75
IBM PC Version ..... $ 95
RS 232 Cable ..... $ 30
8748 family socket adaptor ... $ 98
8751 family socket adaptor ... $174
8755 family socket adaptor ... $135

G7228 Programmer by GTek -- baud to 2400 ... superfast, adaptive programming algorithms ... programs 2764 in one minute.

Programmer $549

Ask us about Gang and PAL programmers.

HEXTRAN Universal HEX File Converter -- Converts to and from Intel, Motorola, MOS Technology, Mostek, RCA, Fairchild, Tektronix, Texas Instruments and Binary formats.

Converter, each version $250

Call Us

If you're thinking about development systems, call us for some straight talk. If we don't have what you need, we'll help you find out who does. If you like, we'll even talk about Bach.

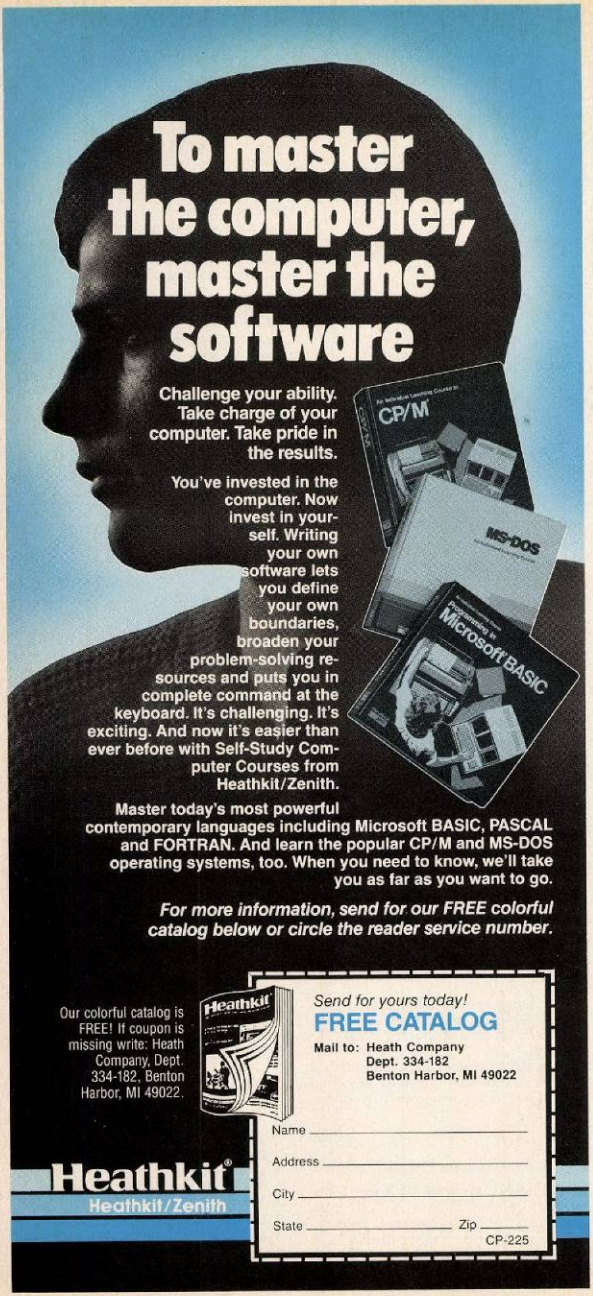
CALL TOLL FREE 1-800-448-8500
(In the U.S. except Alaska and Hawaii)

VISA and Mastercard accepted. All popular disc formats now available -- please specify. Prices do not include shipping and handling -- call for exact quotes. OEM INQUIRIES INVITED.

*Trademark of Digital Research **Trademark of Microsoft

AVOCET SYSTEMS INC.™
DEPT. 684-B
804 SOUTH STATE STREET
DOVER, DELAWARE 19901
302-734-0151 TELEX 467210

Circle 38 on inquiry card.



(text continued from page 204)

four performers commensurately.

Consequently they felt the need to create a new Fraction performer, which they did by placing two Numbers and a Rectangle for the central line on an otherwise empty stage. Since other performers would need to use the values of the numerator and denominator of this Fraction performer, they taught this stage the new cues *getNumerator*, *getDenominator*, and *getValue*. Then they told it to convert itself into a new performer named Fraction and promptly used it in their production.

By the end of the third day, they had a game that worked, that they could respond to, that they liked, and that still needed improvement.

An extra day of work was devoted to adding new features. A Number performer called Parts was added that could be edited by the user; its change action was to show the broom divided into the number of parts indicated. This additional piece of design arose from their interaction with the production; had they been working entirely from a paper sketch, this improvement might not have occurred to them.

They then invited others in our research center to play. Although it had been designed for third-graders, our colleagues found the game interesting and fun to play. They were impressed with the quality of the game and especially with the fact that the designers were nonprogrammers, yet had implemented something so complicated in only a few days.

Eventually we found some children of an appropriate age to be students; they also enjoyed playing the game and spent many hours trying to make a perfect score. Diane now plans to reimplement GoldRush at San Francisco State using the Rehearsal World design as a prototype but changing it to run on different hardware, which might include color and have a different pointing mechanism.

Research Questions

Our experiences with designers have given us confidence that our general ideas about how to make the power of computers accessible to nonprogrammers are correct. We believe that interactive, graphical programs could and

(text continued on page 208)

Circle 155 on inquiry card.

DAISYWHEEL · DOT MATRIX · THERMAL

FUJITSU PRINTERS

You can't buy more Performance for the Price.

Finally, there is a full line of quality printers available to meet a variety of needs. And all from a single manufacturer... FUJITSU. From dependable dot matrix printing to advanced thermal printing, you can't buy more performance for the price.

Quality That's Built In: Fujitsu quality is built into every printer manufactured. That quality translates into high reliability (MTBF), versatile print capability, low maintenance, low noise, and high speeds. And Fujitsu printers are serviced by TRW, a nationwide service organization.

A Complete Printer Line: Fujitsu's dot matrix printer, with its 24 wire head, offers letter quality printing at 80 CPS. With its ability to also produce draft quality, correspondence quality and high resolution graphics, the Fujitsu DPL24 leads dot matrix technology.

In daisy technology, Fujitsu's SP830 is the fastest letter quality printer in the industry at 80 CPS. Fujitsu's SP320 daisywheel printer also provides cost effective letter quality printing at medium speeds. Fujitsu offers thermal printing with its TTP16 printer. The low-cost printer accepts a wide variety of papers and operates quietly at less than 50 dBA.

Call Us Today: Contact Fujitsu America, Inc., at 408-946-8777 for the printer distributor nearest you.

DISTRIBUTORS: *Algoram Computer Products (415) 969-4533, (714) 535-3630, (206) 453-1136, (916) 481-3466; Allen Edwards Associates, Inc. (213) 328-9770, (714) 552-7850, (619) 273-4771, (805) 498-5413; Four Corners Technology (602) 998-4440, (505) 821-5185; Gentry Associates, Inc. (305) 859-7450, (305) 791-8405, (813) 886-0720, (404) 998-2828, (504) 367-3975, (205) 534-9771, (919) 227-3636, (803) 772-6786, (901) 683-8072, (615) 584-0281; Hopkins Associates, Inc. (215) 828-7191, (201) 273-2774; Inland Associates, Inc. (913) 764-7977, (612) 343-3123, (314) 391-6901; Logon, Inc. (201) 646-9222; Lowry Computer Products, Inc. (313) 229-7200, (216) 398-9200, (614) 451-7494, (513) 435-7684, (616) 363-9839, (412) 922-5110, (502) 561-5629; MESA Technology Corp. (301) 948-4350; NACO Electronics Corp. (315) 699-2651, (518) 899-6246, (716) 233-4490; Peak Distributors, Inc., (An affiliate of Dytec/Central) (312) 394-3380, (414) 784-9686, (317) 247-1316, (319) 363-9377; R² Distributing, Inc. (801) 298-2631, (303) 455-5360; S&S Electronics, (617) 458-4100, (802) 658-0000, (203) 878-6800, (800) 243-2776; USDATA (214) 680-9700, (512) 454-3579, (713) 681-0200, (918) 622-8740.*

PERIPHERAL PRODUCTS DIVISION

Quality Lives

FUJITSU

Circle 142 on inquiry card.

Softline

FOR YOUR BOTTOM LINE.

Lotus 1-2-3 $319 | **dBase II $369** | **WordStar Professional $369** | **Multi Mate $279**

WORD PROCESSING/ EDITORS

Easywriter I System (3 pak)	$149
Easywriter II System (3 pak)	$199
Edix/Wordix	$279
Einstein Writer	$199
Final Word	$189
Microsoft Word	$239
Microsoft Word/Mouse	$299
Multimate	$279
PeachText 5000	$199
Perfect Writer/Speller	$249
PFS: Write	$ 95
Samna Word II	$329
Select Word Processor	$199
Spellbinder	$249
SSI Word Perfect	$Call
SuperWriter	$179
Volkswriter	$129
Volkswriter Deluxe	$179
The Word Plus (Oasis)	$109
WordPlus-PC with The Boss	$329
WordStar	$249
WordStar Professional (WS/MM/SS/SI)	$369
WordStar Options Pak (MM/SS/SI)	$189

ACCOUNTING MODULES

Ask Micro Accounting	$299
BPI Accounting	$369
IUS EasyBusiness System	$319
MBA Accounting	$369
Open Systems Accounting	$459
Peachpak 4 (GL/AP/AR)	$239
Peachtree Accounting	$399
Real World Accounting	$469
Star Accounting Partner (GL/AP/AR/PAY)	$269

HOME/PERSONAL FINANCE

Dollars and Sense	$119
Financier II	$119
Home Accountant Plus	$ 99
Tax Preparer 84	$189

DATABASE SYSTEMS

Alpha Data Base Manager II	$179
Condor III	$329
dBase II	$369
DBplus	$ 89
Easy Filer	$219
Friday	$179
InfoStar	$269
Knowledgeman	$309
Perfect Filer	$159
Personal Pearl	$199
PFS: File/PFS: Report	$169
QuickCode	$179
R:base 4000	$299
T/Maker III	$199
TIM IV	$269
Versaform	$249

PROJECT MANAGEMENT

Harvard Project Management	$289
Scitor Project Scheduler	$229
VisiSchedule	$199

GRAPHICS

BPS Business Graphics	$229
Chartman Combo (II&IV)	$349
Chartmaster	$259
dGraph	$189
Fast Graphs	$199
Graphwriter Extended	$429
PC Draw	$219
PFS: Graph	$ 95
VisiTrend/Plot	$199

SPREADSHEETS/MODELING

Jack 2	$Call
Lotus 1-2-3	$319
Multiplan	$159
Perfect Calc	$159
SuperCalc 3	$239
TK! Solver	$Call
VisiCalc IV	$159

LANGUAGES/UTILITIES

Access Manager	$239
Digital Research C Compiler	$219
Display Manager	$299
Microsoft C Compiler	$329
MS Basic Compiler	$249
MS Fortran	$239
Pascal MT+86	$249
Norton Utilities	$ 59

COMMUNICATIONS/ PRODUCTIVITY TOOLS

Crosstalk	$119
Memory Shift	$ 79
Move It	$109
Prokey 3.0	$ 95

HARDWARE PERIPHERALS*

AST Six Pack Plus (64k)	$ 299
Quadboard (0k)	$ 229
Hayes 1200B with Smartcom	$ 439
Hayes Smartmodem 1200	$ 549
Hercules Graphics Board	$ 359
Epson FX-100 Printer	$Call
Comrex II Printer	$Call
NEC 3550 Printer	$1899
C Itoh Prowriter	$ 399
C Itoh Starwriter	$1249

*Add 3% for shipping.

R:base 4000 $299 | **Word & Mouse $299** | **Microsoft C $329** | **AST Six Pak Plus $299**

EXTRA $$$ SAVINGS

With each order, we offer **discount coupons** worth up to **$10** on your next order.

free!

Diskette Library Case

. . . with your order. This attractive case protects, indexes and stores 10 diskettes for quick retrieval. Normally a $10 value, it is now available ***FREE*** to Softline customers

TERMS:
Checks—allow 14 days to clear. Credit processing—add 3%. COD orders—cash, M.O. or certified check—add $3.00. Shipping and handling UPS surface—add $3.00 per item (UPS Blue $6.00 per item). NY State Residents—add applicable sales tax. All prices subject to change.

To Order call

1-800-221-1260

In New York State call **(212) 438-6057**

For technical support and information call (212) 438-6057

Monday thru Friday
9:00 AM - 7:00 PM
Sundays
10:00 AM - 4:00 PM

MasterCard | VISA

Softline

Softline Corporation
3060 Bedford Ave., Brooklyn, N.Y. 11210
TELEX: 421047 ATLN UI

(*text continued from page 206*)

should be built inside an interactive, graphical programming environment. We believe that for such programs, some sort of visual, spatial programming will eventually supplant the current process of writing lines of textual code. Nevertheless, we have many unanswered questions about the nature of visual programming.

An important aspect of the Rehearsal World is that everything is made visible; only things that can be seen can be manipulated. Thus, rather than thinking abstractly, as is necessary in most programming environments, a designer is always thinking concretely, selecting a particular performer, then a particular cue, then observing the cue's instant effect. We know that much of the initial accessibility of the system is due to this concrete, visual, object-oriented approach. What we don't know are its shortcomings.

As designers create increasingly large and sophisticated productions, they may find it a nuisance to have to instantiate everything (even temporary variables) in the form of a performer. There are problems with space on the screen and with visual complexity. Some of these problems are addressed by the ability to collapse a large set of performers into a single new one, which can be made very small while still retaining its original functionality. This helps not only with space but with factoring the production into significant pieces.

While beginning designers benefit from the concreteness, more experienced ones will benefit from being able to think in more general and abstract terms. They are led to think in general terms by the fact that all performers respond to a large set of common cues; they are led to think in abstract terms through the manipulation of Lists and Repeaters. Still, it may be difficult to build productions, for example, that need to access large amounts of data. At some point, the concreteness may become a barrier rather than an advantage.

We know that the "watching" facility is very important to beginners and makes it possible for them to "write" code without learning a language. But it's really very simple and is in no way "programming by example"; it employs

(*text continued on page 210*)

Save Your Memory Before It Blows!

Blackouts...Brownouts...Voltage Surges...Line Noise— They Can Alter Data, Wipe Out RAM Memory, or Damage Equipment.

The Datashield Backup Power Source Can Stand Between Your PC and Disaster from Power Irregularities.

Protect Yourself Four Ways With Datashield

This rechargeable, battery-operated unit — with built-in surge protector — provides maximum protection against all four commercial power problems that can impact your PC: power outage, power drops, voltage spikes, and electromagnetic or radio interference (EMI/RFI).

PC-200. Designed for flexible disc PC's and some hard disc styles.
NOW...ONLY $359.00

XT-300. Designed for most hard disc type models and color monitors.
NOW...ONLY $499.00

Never Say Never...

It can happen to any PC owner. It probably will. Maybe it already did. But it was blamed on something else. Listen:

"Nearly one million Florida homes and businesses lost electricity for 15-30 minutes yesterday morning when a power outage..." **Wall Street Journal**

"Powerline irregularities cause problems for computers...you face hazards every time you plug in a piece of electronic equipment..." **Byte Magazine**

"Powerline associated problems are estimated to cause nearly 70% to 90% of malfunctions in microprocessor-based equipment." **PC Magazine**

"Computer crashes are giving businesses major headaches. The culprit is sudden blackouts..." **U.S. News & World Rep.**

"Fifty percent of our service calls are power related." **Televideo**

"Computer service calls are reduced by 65% when surge protectors are used." **Digital Retail Magazine**

There is one sure way to avoid becoming a statistic: Datashield. The most technologically advanced product in its class. And the most affordable.

What Was Once An Expensive Luxury Is Now An Affordable Necessity

Circle 99 on inquiry card.

Technical Data

Backup Time	
Min. 50% Load	20 Minutes
Min. 100% Load	5 Minutes
Output Rating	
PC-200	200 Watts
XT-300	300 Watts
Typical Transfer Time	
PC-200	4 Milisec. (¼ Cycle)
XT-300	1 Millisec. (1/16 Cycle)
Energy Dissipation	100 Joules

Datashield®

Datashield ® is a U.S. registered trademark of PTI Industries

320 River Street, Santa Cruz, California 95060 (408) 429-6881

(text continued from page 208)

no generalizations but merely makes a textual record of a performer being sent a cue, perhaps with parameters. Again, advanced designers might be led to think abstractly rather than specifically if the Rehearsal World provided a more powerful watching facility that was capable of some form of generalization.

In the Rehearsal World, button action and change action are the major mechanisms for expressing the interactions of all performers; a few performers, like the Repeater, the List, and the Traveler, have other special actions as well. Designers find these actions very natural and so far have had no difficulty describing their needs in these terms. However, the Rehearsal World does not provide designers with the facility to create new types of actions for new performers, and this may become a problem in the future.

The Rehearsal World supports multiple processes in such a natural way that our designers are not surprised by the existence of this facility as they interrupt whatever they're doing to do something else. However, we have little experience with designers using multiple processes in some production and expect a variety of conceptual and mechanical difficulties to arise.

Designers express actions in a procedural fashion, instructing a performer to send a cue under certain conditions. We are curious about how designers would deal with a constraint-based Rehearsal World in which the relationships between performers were expressed in terms of conditions that should always hold true (for example, that the value of a Number should always be twice that of another Number). We hope that researchers working on similar design environments will explore these questions. ■

REFERENCES

1. Brown, Dean, and Joan Lewis. "The Process of Conceptualization." Educational Policy Center Research Note EPRC-6747-9. SRI Project 6747. December, 1968.
2. Oettinger, Anthony, with Sema Marks. *Run, Computer, Run*. Cambridge, MA: Harvard University Press, 1969.
3. Gould, Laura, and William Finzer. "A Study of TRIP: A Computer System for Animating Time-Rate-Distance Problems." *International Journal of Man-Machine Studies* (1982) 17, 109–126.
4. 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, AZ: 1978.
5. BYTE, August 1981.
6. Goldberg, Adele. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, 1984.
7. Goldberg, Adele, and David Robson. *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.
8. Krasner, Glenn, ed. *Smalltalk-80, Bits of History, Words of Advice*. Reading, MA: Addison-Wesley, 1983.

DEALERS ONLY!

IBM PC™ Compatible SANYO

Business Computer System

STANDARD FEATURES:
- MS DOS
- 16 Bit 8088 CPU
- SANYO BASIC
- 128K Internal Memory
- Centronics Printer Port
- Color Graphic Capabilities
- Diagnostics, Utilities, Speaker & Joystick Port

CALL FOR PRICING!

MBC 550

- 1 single sided/double density disk drive (160K)
- WordStar & CalcStar Included

RETAIL-$999.00

MBC 555

- 2 single sided/double density disk drives (320K)
- 128K memory expandable to 256K
- WordStar, CalcStar, DataStar, InfoStar, & MailMerge Included

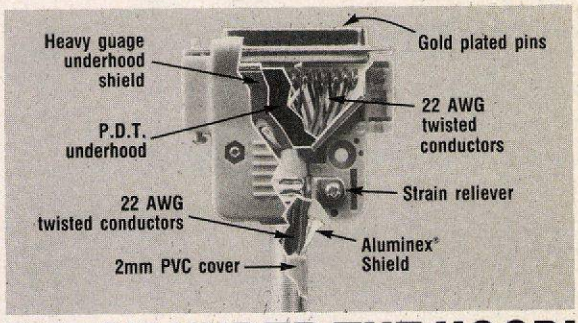
Also available in double sided drives

MEC MICRO·EQUIP·CORP

MICRO EQUIPMENT CORPORATION
245 West Wieuca Rd.
Suite 210
Atlanta, Ga. 30342
404/843-3128

Circle 389 on inquiry card.

BEFORE YOU BUY CABLE ASSEMBLIES,



CHECK UNDER THE HOOD!

DATA SPEC™ cable assemblies are the very best. Each cable is fully shielded to exceed FCC EMI/RFI emission requirements. Furthermore, the unique P.D.T. technique is employed beneath the hood shield for maximum integrity under the most adverse conditions. DATA SPEC™ was the first to use the P.D.T. process, and cable assemblies constructed with P.D.T. carry a lifetime warranty. DATA SPEC™ has interface cables for all your requirements: Modems, Monitors, Disk Drives, and much more. Insist on DATA SPEC™ cables in the bright orange package. Available at better computer dealers everywhere. For more information, call or write:

DATA SPEC™
A Division of Alliance Research Corporation
18215 Parthenia Street, Northridge, CA 91325 (818) 701-5853

Circle 97 for Dealer inquiries.
Circle 98 for End-User inquiries.

DATAMATION®

POWER TOOLS FOR PROGRAMMERS

by Beau Sheil

Reprinted with permission of Datamation® Magazine, © copyright by Technical Publishing, a Dun & Bradstreet Company, 1983—all rights reserved.

Emerging from AI labs, exploratory programming environments can handle complex, interactive applications that structured methods box into a corner.

POWER TOOLS FOR PROGRAMMERS

by Beau Sheil

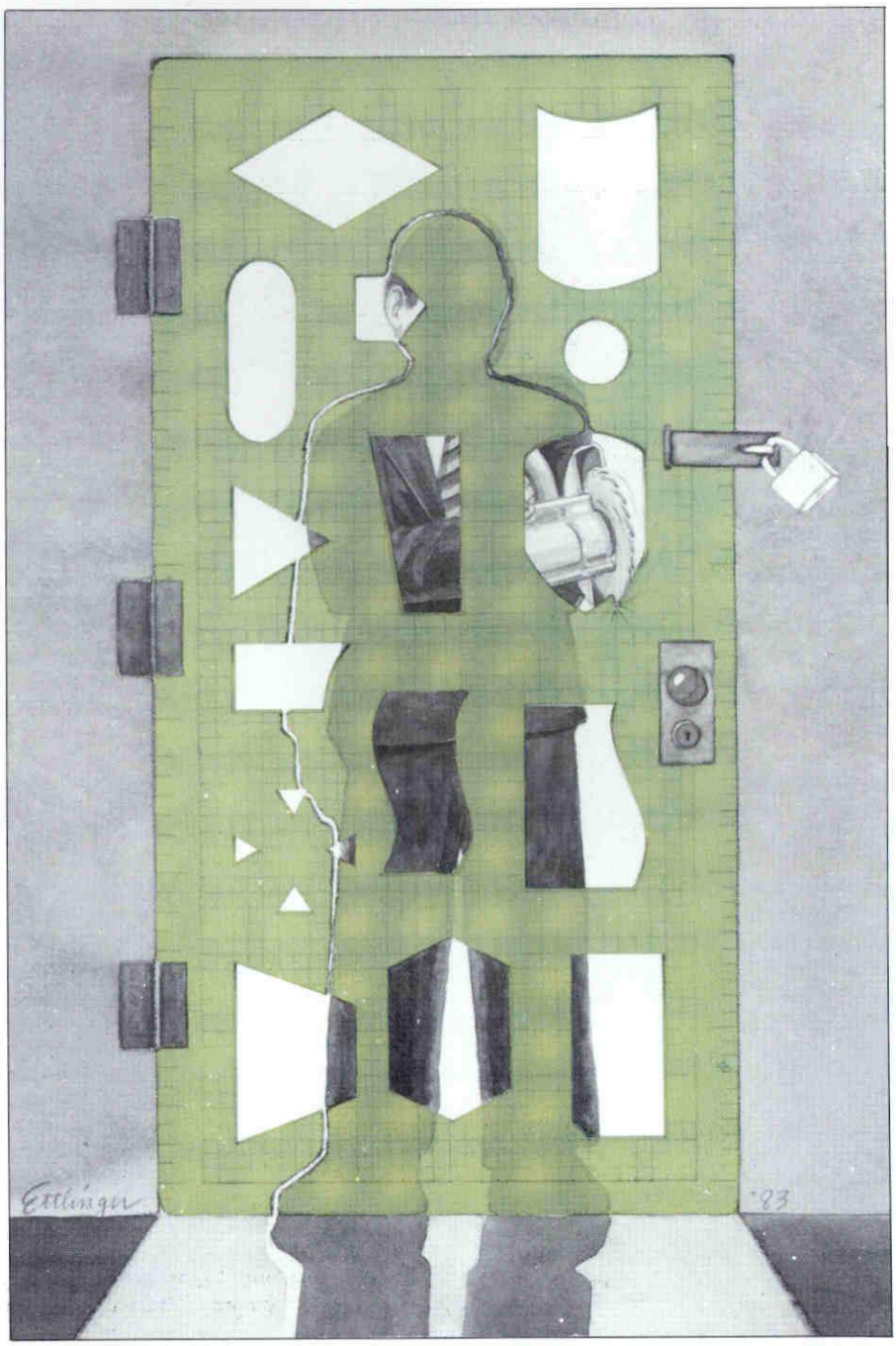


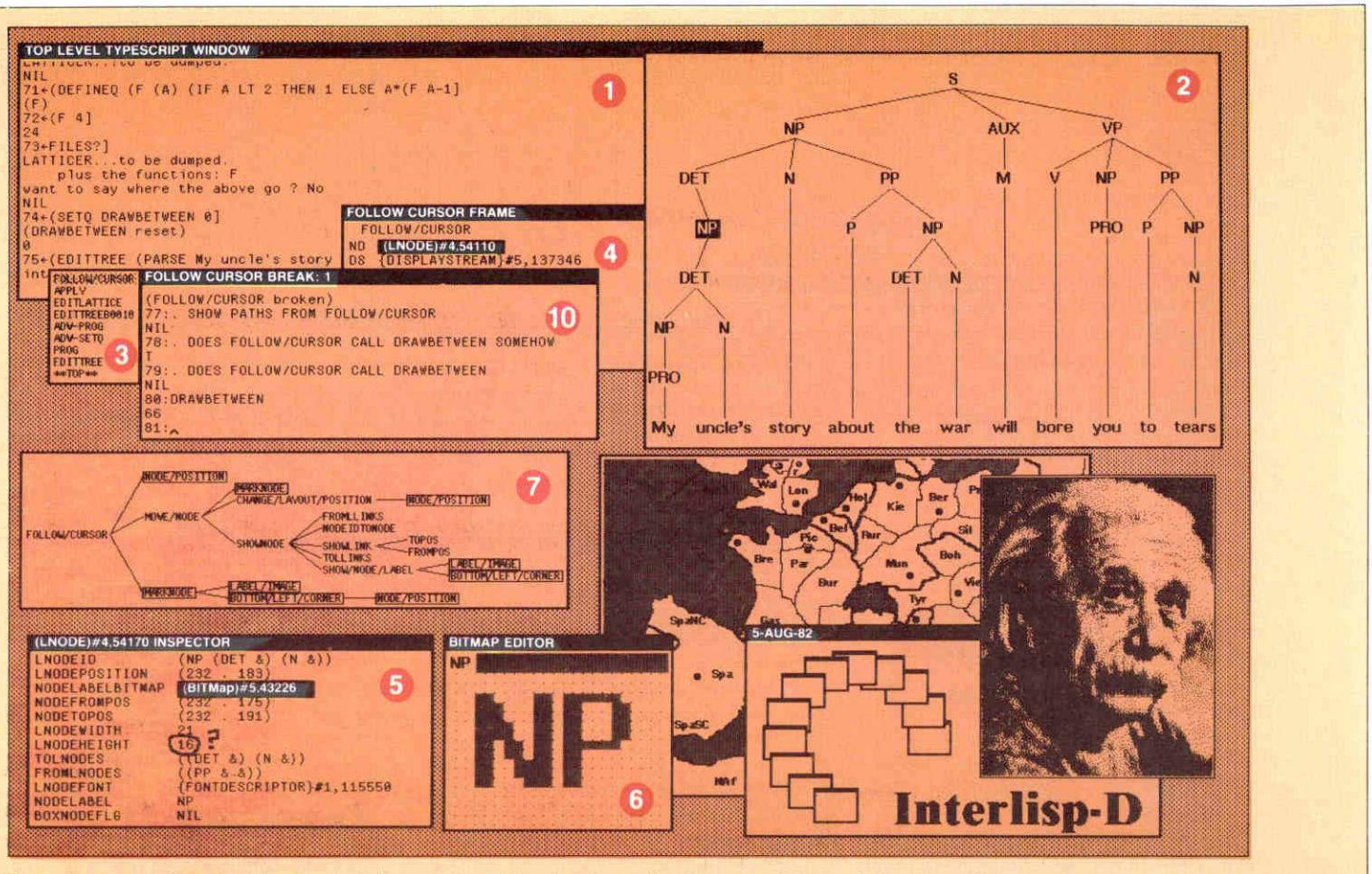
ILLUSTRATION BY DORIS ETTINGER

An oil company needs a system to monitor and control the increasingly complex and frequently changing equipment used to operate an oil well. An electronic circuit designer plans to augment a circuit layout program to incorporate a variety of vaguely stated design rules. A newspaper wants a page layout system to assist editors in balancing the interlocking constraints that govern the placement of stories and advertisements. A government agency envisions a personal workstation that would provide a single integrated interface to a variety of large, evolving database systems.

Applications like these are forcing the commercial deployment of a radically new kind of programming system. First developed to support research in artificial intelligence and interactive graphics, these new tools and techniques are based on the notion of exploratory programming, the conscious intertwining of system design and implementation. Fueled by dramatic changes in the cost of computing, such exploratory programming environments have become a commercial reality virtually overnight. No fewer than four such systems were displayed at NCC '82 and their numbers are likely to increase rapidly as their power and range of application become more widely appreciated.

Despite the diversity of subject matter, a common thread runs through our example applications. They are, of course, all large, complex programs whose implementations will require significant resources. Their more interesting similarity, however, is that it is extremely difficult to give complete specifications for any of them. The reasons range from sheer complexity (the circuit designer can't anticipate all the ways in which his design rules will interact), through continually changing requirements (the equipment in the oil rig changes, as do the information bases that the government department is required to consult), to the subtle human factors issues that determine the effectiveness of an interactive graphics interface.

Whatever the cause, a large programming project with uncertain or changing specifications is a particularly deadly combi-



These two screen images show some of the exploratory programming tools provided in the Xerox Interlisp-D programming environment. The screen is divided into a series of rectangular areas or windows, each of which provides a view onto some data or process, and which can be reshaped and repositioned at will by the user. When they overlap, the occluded portion of the lower window is automatically saved, so that it can be restored when the overlapping window is removed. Since the display is bitmapped, each window can contain an arbitrary mixture of text, lines, curves, and pictures composed of half-tones or solids. The image of Einstein, for instance, was produced by scanning a photograph and storing it digitally.

In the typescript window (labeled 1), the user has defined a program F (factorial) and has then immediately run it, giving an input of 4 and getting a result of 24. Next, in the same window, he queries the state of his files, finding that one file (LATTICER) has already been changed and one function (F) has been defined but not associated with any file yet. The user sets the value of DRAWBETWEEN to 0 in command 74, and the system notes that this is a change and adds DRAWBETWEEN to the set of "changed objects" that might need to be saved.

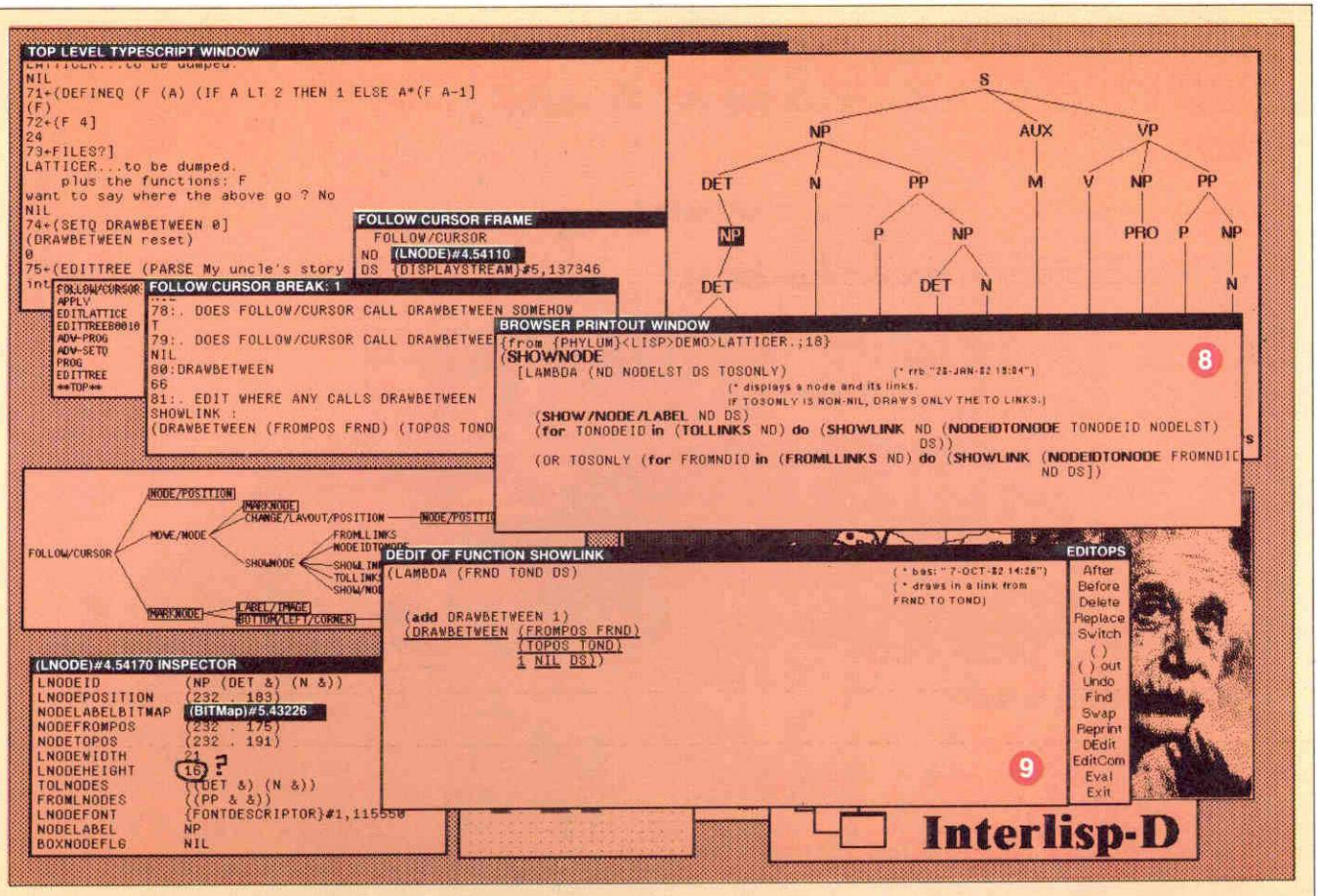
Then, the user runs the program EDITTREE, giving it a parse tree for the sentence "My uncle's story about the war will bore you to tears." This opens up the big window (2) on the right in which the sentence diagram is drawn. Using the mouse, the user starts to move the NP node on the left (which is inverted to show that it is being moved).

While the move is taking place, the user interrupts the tree editor, which suspends the computation and causes three "break" windows to appear on top of the lower edge of the typescript. The smallest window (3) shows the dynamic state of the computation, which has been broken inside a subprogram called FOLLOW/CURSOR. The "FOLLOW/CURSOR Frame" window (4) to the right shows the value of the local variables bound by FOLLOW/CURSOR. One of them has been selected (and so appears inverted) and in response, its value has been shown in more detail in the window (5) at the lower left of the screen. The user has marked one of the component values as suspicious by circling it using the mouse. In addition, he has asked to examine the contents of the BITMAP component, which has

nation for conventional programming techniques. Virtually all modern programming methodology is predicated on the assumption that a programming project is fundamentally a problem of implementation, rather than design. The design is supposed to be decided on first, based on specifications provided by the client; the implementation follows. This dichotomy is so important that it is standard practice to recognize that a client may have only a partial understanding of his needs, so that extensive consultations may be required to ensure a complete specification with which the client will remain happy. This dialog guarantees a fixed specification that will form a stable base for an implementation.

The vast bulk of existing programming practice and technology, such as structured design methodology, is designed to ensure that the implementation does, in fact, follow the specification in a controlled fashion, rathern than wander off in some unpredictable direction. And for good reason. Modern programming methodology is a significant achievement that has played a major role in preventing the kind of implementation disasters that often befell large programming projects in the 1960s.

The implementation disasters of the 1960s, however, are slowly being succeeded by the design disasters of the 1980s. The projects described above simply will not yield to conventional methods. Any attempt to obtain an exact specification from the client is bound to fail because, as we have seen, the client does not know and cannot anticipate exactly what is required. Indeed, the most striking thing about these examples is that the clients' statements of their problems are really aspirations, rather than specifications. And since the client has no experience on which to ground these aspirations, it is only by exploring the properties of some putative solutions that the client will find out what is really needed. No amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works.



opened up a bitmap edit window (6) to the right. This shows an enlarged copy of the actual NP image that is being moved by the tree editor. Then, inside the largest of the three break windows (10) the user has asked some questions about the FOLLOW/CURSOR subprogram that was running when he interrupted, and queried the value of DRAWBETWEEN (now 66). The SHOW PATHS command brought up the horizontal tree diagram on the left (7), which shows which subprograms call each other, starting at FOLLOW/CURSOR.

Each node in the call tree produced by the SHOW PATHS command is an active element that will respond to the user's selecting it with the mouse. In the second image, the user has selected the SHOWNODE subprogram, which has caused its source code to be retrieved from the file (‹LISP›DEMO›LATTICER) on the remote file server (PHYLUM) where it was stored, and displayed in the "Browser printout window" (8) which has been opened at middle right. User functions and extended Lisp forms (like *for* and *do*) are highlighted by system-generated font changes.

By selecting nodes in the SHOW PATHS window, the user could also have edited the code or obtained a summary description of any of its subprograms.

Instead, the user has asked (in the break typescript window (10)) to edit wherever anybody calls the DRAWBETWEEN system primitive (which draws lines between two specified points). This request causes the system to consult its dynamically maintained database of information about user programs, wherein it finds that the subprogram SHOWLINK calls DRAWBETWEEN. It therefore loads the code for SHOWLINK into an edit window (9) that appears under the "Browser printout window." The system then automatically finds and underlines the first (and only) call on DRAWBETWEEN. Note that on the previous line DRAWBETWEEN is used as a variable (the same variable the user set and interrogated earlier). The system, however, knows that this is not a subprogram call, so it has been skipped over. If the user were to make any change to this subprogram in the editor, not only would the change take effect immediately, but SHOWLINK would be marked as needing to be updated in its file and the information about it in the subprogram database would be updated. This, in turn, would cause the SHOW PATHS window to be repainted, as its display might no longer be valid.

The consequences of approaching problems like these as routine implementation exercises are dramatic. First, the implementation team begins by pushing for an exact specification. How long the client resists this coercion depends on how well he really understands the limits of his own grasp of the problem. Sooner or later, however, with more or less ill-feeling, the client accepts a specification and the implementation team goes to work.

The implementors take the specification, partition it, define a module structure that reflects this partitioning, freeze the interfaces between them, and repeat this process until the problem has been divided into a large number of small, easily understandable, and easily implementable pieces. Control over the implementation process is achieved by the imposition of structure, which is then enforced by a variety of management practices and programming tools.

USE OF INTERNAL RIGIDITY

Since the specification, and therefore the module structuring, is considered fixed, one of the most effective methods for enforcing it is the use of redundant descriptions and consistency checking. Hence the importance of techniques such as interface descriptions and static type checking, which require that multiple statements of various aspects of the design be included in the program text. These statements allow mechanical checks that ensure that each piece of the system remains consistent with the rest. In a well-executed conventional implementation project, a great deal of internal rigidity is built into the system, ensuring its orderly development.

The problems usually emerge at system acceptance time, when the client requests not just superficial, but radical changes, either as a result of examining the system or for some completely exogenous reason. From the point of view of conventional programming practice, this indicates a failure at specification time. The software engineer

The implementation disasters of the 1960s are slowly being succeeded by the design disasters of the 1980s.

should have been more persistent in obtaining a fuller description of the problem, in involving all the affected parties, etc. This is often true. Many ordinary implementation exercises are brought to ruin because the consequences of the specification were never fully agreed upon. But that's not the problem here. The oil company couldn't anticipate the addition of a piece of equipment quite different from the device on which the specification was based. No one knew that the layout editors would complain that it doesn't "feel right" now that they can no longer physically handle the copy (even in retrospect, it's unclear why they feel that way and what to do about it), etc., etc., etc. Nor would any amount of speculation by either client or software engineer have helped. Rather, it would have just prompted an already nervous client to demand whole dimensions of flexibility that would not in fact be needed, leaving the system just as unprepared for the ones that eventually turned out to matter.

Whatever the cause, the implementation team has to rework the system to satisfy a new, and significantly different, specification. That puts them in a situation that conventional programming methodology simply refuses to acknowledge—except as something to avoid. As a result, their programming tools and methods are suddenly of limited effectiveness. The redundant descriptions and imposed structure that were so effective in constraining the program to follow the old specification have lost none of their efficacy—they still constrain the program to follow the old specification. And they're difficult to change. The whole point of redundancy is to protect the design from a single unintentional change. But it's equally well protected against a single intentional change. Thus, all the changes have to be made everywhere. (Since this should never happen, there's no methodology to guide or programming tools to assist this process.) Of course, if the change is small (as it "should" be), there is no particular problem. But if it is large enough to cut across the module structure, the implementation team finds that it has to fight its way out of its previous design.

Still no major problem, if that's the end of the matter. But it rarely is. The new system will suggest yet another change. And so on. After a few iterations of this, not only are the client and the implementation team not on speaking terms, but the repeated assaults on the module structure have likely left it looking like spaghetti. It still gets in the way (fire walls are just as impenetrable if laid out at random as they are when laid out straight), but has long ceased to be of any use to anyone except to remind them of the project's sorry history. Increasingly, it is actively subverted (enter LOOPHOLES, UNSPECS, etc.) by programmers whose patience is running thin. Even if the design were suddenly to stabilize (unlikely in the present atmosphere), all the seeds have now been sown for an implementation disaster as well.

EXPLORE DESIGN PROBLEMS

The alternative to this kind of predictable disaster is not to abandon structured design for programming projects that are, or can be made to be, well defined. That would be a tremendous step backwards. Instead, we should recognize that some applications are best thought of as design problems, rather than implementation projects. These problems require programming systems that allow the design to emerge from experimentation with the program, so that design and program develop together. Environments in which this is possible were first developed in artificial intelligence and computer graphics, two research areas that are particularly prone to specification instability.

At first sight, artificial intelligence might seem an unlikely source of programming methodology. But constructing programs, in particular programs that carry out some intelligent activity, is central to artificial intelligence. Since almost any intelligent activity is likely to be poorly understood (once a program becomes well understood we usually cease to consider it "intelligent"), the artificial intelligence programmer invariably has to restructure his program many, many times before it becomes reasonably proficient. In addition, since intelligent activities are complex, the programs tend to be very large, yet they are invariably built by very small teams, often a single researcher. Consequently, they are usually at or beyond the manageable limits of complexity for their implementors. In response, a variety of programming environments based on the Lisp programming language have evolved to aid in the development of these large, rapidly changing systems.

The rapidly developing area of interactive graphics has encountered similar problems. Fueled by the swift drop in the cost of computers capable of supporting interactive graphics, there has been an equally swift development of applications that make heavy use of interactive graphics in their user interfaces. Not only was the design of such interfaces almost completely virgin territory as recently as 10 years ago, but even now, when there are a variety of known techniques (menus, windows, etc.) for exploiting this power, it is still very difficult to determine how easy it will be to use a proposed user interface and how well it will match the user's needs and expectations in particular situations. Consequently, complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so.

While interface design has always required some amount of tuning, the vastly increased range of possibilities available in a full graphics system has made the design space unmanageably large to explore without extensive experimentation. In response, a variety of systems, of which Smalltalk is the best known, have been developed to facilitate this experimentation by providing a wide range of built-in graphical abstractions and methods of modifying and combining them together into new forms.

In contrast to conventional programming technology, which restrains the programmer in the interests of orderly development, exploratory programming systems must amplify the programmer in the interests of maximizing his effectiveness. Exploration in the realm of programming can require small numbers of programmers to make essentially arbitrary transformations to very large amounts of code. Such programmers need programming power tools of considerable capacity or they will simply be buried in detail. So, like an amplifier, their programming system must magnify their necessarily limited energy and minimize extraneous activities that would otherwise compete for their attention.

SOURCES OF DESIGN POWER

One source of such power is the use of interactive graphics. Exploratory programming systems have capitalized on recent developments in personal computing with extraordinary speed. The Xerox 1108 Interlisp-D system, for example, uses a large format display and a "mouse" pointing device to allow very high bandwidth communication with the user. Designers of exploratory programming environments have been quick to seize on the power of this combination to provide novel programming tools, as we shall see.

In addition to programming tools, these personal machine environments allow the standard features of a professional workstation, such as text editing, file management, and electronic mail, to be provided within the programming environment itself. Not only are these facilities just as effective in enhancing the productivity of programmers as they are for other professionals, but their integration into the programming environment allows them to be used at any time during programming. Thus, a programmer who has encountered a bug can send a message reporting it while remaining within the debugger, perhaps including in the message some information, like a back-trace, obtained from the dynamic context.

Another source of power is to build the important abstract operations and objects

Redundancy protects the design from unintentional change—but it's equally well protected against intentional change.

of some given application area directly into the exploratory environment. All programming systems do this to a certain extent; some have remarkably rich structures for certain domains, (e.g., the graphics abstractions embedded within Smalltalk). If the abstractions are well chosen, this approach can yield a powerful environment for exploration within the chosen area, because the programmer can operate entirely in substantively meaningful abstractions, taking advantage of the considerable amount of implementation and design effort that they represent.

The limitations of this approach, however, are clear. Substantive abstractions are necessarily effective only within a particular topic area. Even for a given area, there is generally more than one productive way to partition it. Embedding one set of abstractions into the programming system encourages developments that fit within that view of the world at the expense of others. Further, if one enlarges one's area of activity even slightly, a set of abstractions that was once very effective may become much less so. In that situation, unless there are effective mechanisms for reshaping the built-in abstractions to suit the changed domain, users are apt to persist with them, at the cost of distorting their programs. Embedded abstractions, useful though they are, by themselves enable only exploration in the small, confined within the safe borders where the abstractions are known to be effective. For exploration in the large, a more general source of programming power is needed.

Of course, the exact mechanisms that different exploratory systems propose as essential sources of programming power vary widely, and these differences are hotly debated within their respective communities. Nevertheless, despite strong surface differences, these systems share some unusual characteristics at both the language and environment level.

THE LANGUAGE LEVEL

The key property of the programming languages used in exploratory programming systems is their emphasis on minimizing and deferring the constraints placed on the programmer, in the interests of minimizing and deferring the cost of making large-scale program changes. Thus, not only are the conventional structuring mechanisms based on redundancy not used, but the languages make extensive use of late binding, i.e., allowing the programmer to defer commitments for as long as possible.

The clearest example is that exploratory environments invariably provide dynamic storage allocation with automatic reclamation (garbage collection). To do otherwise imposes an intolerable burden on the programmer to keep track of all the paths through his program that might access a particular piece of storage to ensure that none of them access or release it prematurely (and that someone does release it eventually!). This can only be done by careful isolation of storage management or with considerable administrative effort. Both are incompatible with rapid, unplanned development, so neither is acceptable. Storage management must be provided by the environment itself.

Other examples of late binding include the dynamic typing of variables (associating data type information with a variable at run-time, rather than in the program text) and the dynamic binding of procedures. The freedom to defer deciding the type of a value until run-time is important because it allows the programmer to experiment with the type structure itself. Usually, the first few drafts of an exploratory program implement most data structures using general, inefficient structures such as linked lists discriminated (when necessary) on the basis of their contents. As experience with the application evolves, the critical distinctions that determine the type structure are themselves determined by experimentation, and may be among the last, rather than the first, decisions to evolve. Dynamic typing makes it easy for the programmer to write code that keeps these decisions as tacit as possible.

The dynamic binding of procedures entails more than simply linking them at load-time. It allows the programmer to change dynamically the subprocedures invoked by a given piece of code, simply by changing the run-time context. The simplest form of this is to allow procedures to be used as arguments or as the value of variables. More sophisticated mechanisms allow procedure values to be computed or even encapsulated inside the data values on which they are to operate. This packaging of data and procedures into a single object, known as object-oriented programming, is a very powerful technique. For example, it provides an elegant, modular solution to the problem of generic procedures (i.e., every data object can be thought of as providing its own definition for common actions, such as printing, which can be invoked in a standard way by other procedures). For these reasons, object-oriented programming is a widely used exploratory programming technique and actually forms the basic programming construct of the Smalltalk language.

The dynamic binding of procedures can be taken one step further when procedures are represented as data structures that can be effectively manipulated by other programs. While this is of course possible to a limited extent by reading and writing the text of program source files, it is of much greater significance in systems that define an explicit representation for programs as syntax trees or their equivalent. This, coupled with the interpreter or incremental compiler provided by most exploratory programming systems, is an extraordinarily powerful tool. Its most dramatic application is in programs that construct other programs, which they later invoke. This technique is often used in artificial intelligence in situations where the range of possible behaviors is too large to encode efficiently as data structures but can easily be expressed as combinations of procedure fragments. An example might be a system that "understands" instructions given in natural language by analyzing each input as it is received, building a program that captures its meaning, and then evaluating that program to achieve the requested effect.

A BASIC TECHNIQUE EXPANDED

Aside from such specialized applications, effective methods for mechanically manipulating procedures enable two other significant developments. The first is the technique of program development by writing interpreters for special purpose languages. Once again, this is a basic technique of artificial intelligence that has much wider applicability. The key idea is that one develops an application by designing a special language in which the application is relatively easy to state. Like any notation, such a language provides a concise representation that suppresses common or uninteresting features in favor of whatever the designer decides is more important.

A simple example is the use of notations like context-free grammars (BNF) to "metaprogram" the parsers for programming languages. Similar techniques can be used to describe, among other things, user interfaces, transaction sequences, and data transformations. Application development in this framework is a dialectic process of designing the application language and developing an interpreter for it, since both the language and the interpreter will evolve during development. The simplest way of doing this is to evolve the application language out of the base provided by the development language. Simply by allowing the application language interpreter to call the development language interpreter, expressions from the development language can be used wherever the application language currently has insufficient power. As one's understanding of the problem develops, the application language becomes increasingly powerful and the need to escape into the development language becomes less important.

The other result of having procedures that are easily manipulated by other procedures is that it becomes easy to write program manipulation subsystems. This in turn has two key consequences. First, the exploratory

Conventional programming technology restrains the programmer; exploratory systems amplify him.

programming language itself can grow. The remarkable longevity of Lisp in the artificial intelligence community is in large part due to the language having been repeatedly extended to include modern programming language syntax and constructions. The vast majority of these extensions were accomplished by defining source-to-source transformations that converted new constructions into more conventional Lisp. The ease with which this can be done allows each user, and even each project, to extend the language to capture the idioms that are found to be locally useful.

Second, the accessibility of procedures to mechanical manipulation facilitates the development of programming support tools. All exploratory programming environments boast a dazzling profusion of programming tools. To some extent, this is a virtue of necessity, as the flexibility necessary for exploration has been gained at considerable sacrifice in the ability to impose structure. That loss of structure could easily result in a commensurate loss of control by the programmer. The programming tools of the exploratory environment enable the programmer to reimpose the control that would be provided by structure in conventional practice.

Programming tools achieve their effectiveness in two quite different ways. Some tools are simply effective viewers into the user's program and its state. Such tools permit one to find information quickly, display it effectively, and modify it easily. A wide variety of tools of this form can be seen in the two Interlisp-D screen images (see box, p. 132), including data value inspectors (which allow a user to look at and modify the internal structure of an object), editors for code and data objects, and a variety of break and tracing packages. Especially when coupled with a high bandwidth display, such viewers are very effective programming tools.

A WIDE VARIETY OF TOOLS

The other type of programming tool is knowledge based. Viewer-based tools, such as a program text editor, can operate effectively with a very limited understanding of the material with which they deal. By contrast, knowledge-based tools must know a significant amount about the content of a user's program and the context in which it operates. Even a very shallow analysis of a set of programs (e.g., which programs call which other ones) can support a variety of effective programming tools. A program browser allows a programmer to track the various dependencies between different parts of a program by presenting easy to read summaries that can be further expanded interactively.

Deeper analysis allows more sophisticated facilities. The Interlisp program analyzer (Masterscope) has a sufficiently detailed knowledge of Lisp programs that it can provide a complete static analysis of an arbitrary Lisp program. A wide variety of tools have been constructed that use the database provided by this analysis to answer complex queries (which may require significant reasoning, such as computing the transitive closure of some property), to make systematic changes under program control (such as making some transformation wherever a specified set of properties hold), or to check for a variety of inconsistent usage errors.

Finally, integrated tools provide yet another level of power. The Interlisp system notices whenever a program fragment is changed (by the editor or by redefinition). The program analyzer is then informed that any existing analysis is invalid, so that incorrect answers are not given on the basis of old information. The same mechanism is used to notify the program management subsystem (and eventually the user, at session end) that the corresponding file needs to be updated. In addition, the system will remember the previous state of the program, so that at any subsequent time the programmer can undo the change and retreat (in which case, of course, all the dependent changes and notifications will also be undone). This level of cooperation between tools not only provides immense power to the programmer, but relieves him of detail that he would otherwise have to manage himself. The result is that more attention can be paid to exploring the design.

A key, but often neglected, component of an exploratory programming system is a set of facilities for program contraction. The development of a true exploratory program is design limited, so that is where the effort has to go. Consequently, the program is often both inefficient and inelegant when it first achieves functional acceptability. If the exploration were an end in itself, this might be of limited concern. However, it is more often the case that a program developed in an exploratory fashion must eventually be used in some real situation. Sometimes, the time required to reimplement (using the prototype program as a specification) is prohibitive. Other times, the choice of an exploratory system was made to allow for expected future upheaval, so it is essential to preserve design flexibility. In either event, it is necessary to be able to take the functionally adequate program and transform it into one whose efficiency is comparable to the best program one could have written, in any language, had only one known what one was doing at the outset.

The importance of being able to make this post hoc optimization cannot be overemphasized. Without it, one's exploratory programs will always be considered toys; the pressure to abandon the exploratory environment and start implementing in a real one will be overwhelming; and, once that move is made (and it is always made too soon), exploration will come to an end. The requirement for efficient implementation places two burdens on an exploratory programming system. First, the architecture must allow an efficient implementation. For example, the obligatory automatic storage allocation mechanism must either be so efficient that its overhead is negligible, or it must permit the user to circumvent it (e.g., to allocate storage statically) when and where the design has stabilized enough to make this optimization possible.

Second, as the performance engineering of a large system is almost as difficult as its initial construction, the environment must provide performance engineering tools, just as it provides design tools. These include good instrumentation, a first-class optimizing compiler, program manipulation tools (including, at the very least, full functionality compiler macros), and the ability to add declarative information where necessary to guide the program transformation. Note that, usually, performance engineering takes place not as a single "post-functionality optimization phase," but as a continuous activity throughout the development, as different parts of the system reach design stability and are observed to be performance critical. This is the method of progressive constraint, the incremental addition of constraints as and when they are discovered and found important, and is a key methodology for exploratory development.

Both of these concerns can be most clearly seen in the various Lisp-based systems. While, like all exploratory environments, they are often used to write code very quickly without any concern for efficiency, they are also used to write artificial intelligence programs whose applications to real problems are very large computations. Thus, the ability to make these programs efficient has long been of concern, because without it they would never be run on any interesting problems.

More recently, the architectures of the new, personal Lisp machines like the 1108 have enabled fast techniques for many of the operations that are relatively slow in a traditional implementation. Systems like Interlisp-D, which is implemented entirely in Lisp, including all of the performance-critical system code such as the operating system, display software, device handlers, etc., show the level of efficiency that is now possible within an exploratory language.

The increasing importance of applications that are very poorly understood, both by their clients and by their would-be implementors, will make exploratory development a key technique for the 1980s. Radical changes in the cost of computing power have already made such systems cost-effective vehicles for

The programming languages used in exploratory systems minimize and defer constraints on the programmer.

the delivery of application systems in many areas. As recently as five years ago, the tools and language features we have discussed required the computational power of a large mainframe costing about $500,000. Two years ago, equivalent facilities became available on a personal machine for about $100,000, and a year later, about $50,000. Now, a full-scale exploratory development system can be had for about $25,000. For many applications, the incremental cost has become so small over that required to support conventional technology that the benefits of exploratory development (and redevelopment!) are now decisive.

One consequence of this revolutionary change in the cost-effectiveness of exploratory systems is that our idea of exploratory problems is going to change. Exploratory programming was developed originally in contexts where change was the dominant factor.

There is, however, clearly a spectrum of specification instability. Traditionally, the cost of exploratory programming systems, both in terms of the computing power required and the run-time inefficiencies incurred, confined their use to only the most volatile applications. Thus, the spectrum was arbitrarily dichotomized into exploratory (very few) and standard (the vast majority). Unfortunately, the reality is that unexpected change is far more common in standard applications than we have been willing to admit. Conventional programming techniques strive to preserve a stability that is only too often a fiction. Since exploratory programming systems provide tools that are better adapted to this uncertainty, many applications that are now being treated as standard but which in fact seem to require moderate levels of ongoing experimentation may turn out to be more effectively developed in an exploratory environment.

We can also expect to see a slow infusion of exploratory development techniques into conventional practice. Many of the programming tools of an exploratory programming system (in particular, the information gathering and viewing tools) do not depend on the more exploratory attributes of either language or environment and could thus be adapted to support programming in conventional languages like FORTRAN and COBOL. Along with these tools will come the seeds of the exploratory perspective on language and system design, which will gradually be incorporated into existing programming languages and systems, loosening some of the bonds with which these systems so needlessly restrict the programmer.

To those accustomed to the precise, structured methods of conventional system development, exploratory development techniques may seem messy, inelegant, and unsatisfying. But it's a question of congruence: precision and inflexibility may be just as disfunctional in novel, uncertain situations as sloppiness and vacillation are in familiar, well-defined ones. Those who admire the massive, rigid bone structures of dinosaurs should remember that jellyfish still enjoy their very secure ecological niche. ✱

Beau Sheil is on the research staff at the Palo Alto Research Center of the Xerox Corp., where he has been since receiving his PhD in computer science from Harvard University in 1976. His research interests include programming systems and the psychology of programming. Many of these ideas were first developed, and later polished, in discussions with John Seely Brown and other colleagues in cognitive and instructional sciences at Xerox PARC.

DATAMATION®

Smalltalk Grows Up

Thanks to a boost from IBM and Microsoft, and a growing set of support tools, Smalltalk is finally beginning to sound good to IS.

BY JEFF MOAD

If object-oriented programming (OOP) technologies are the wave of the future, why hasn't Smalltalk, the granddaddy of OOP languages, been more successful?

After all, Smalltalk, which was developed at the Xerox PARC research labs in Palo Alto in the mid-1970s, was one of the first languages to reject the operator/operand, linear style of more conventional programming languages. Instead, Smalltalk uses self-contained data structures called objects, which programmers can combine and reuse in their applications. Its proponents said Smalltalk would significantly improve programmer productivity and make long-term program maintenance and enhancement easier.

But Smalltalk never really took off. IS applications development managers have tended to view it as a fringe language, weak on performance and lacking critical support features, such as stable database interfaces and well-established development methodologies. Thus, IS managers have been reluctant to retrain their COBOL programmers to use Smalltalk. ParcPlace Systems, a spin-off from Xerox PARC, and a few small consulting firms were pretty much alone in trying to convince IS to take Smalltalk seriously.

A Small Surge

All that may be changing, however. A number of IS organizations are moving beyond experimenting or prototyping applications with Smalltalk and are beginning to develop critical applications. Several small vendors have recently entered the Smalltalk market with products that make the language easier to use and more productive for large applications. And a ParcPlace competitor has even emerged. Digitalk Inc. of Los Angeles is now selling lower cost versions of Smalltalk targeted at OS/2 and Windows users.

Smalltalk is not about to replace COBOL, but it is finally maturing into a viable choice in application development, especially for users looking for a tool to speed development of advanced graphical user interfaces (GUIs) in client/server applications.

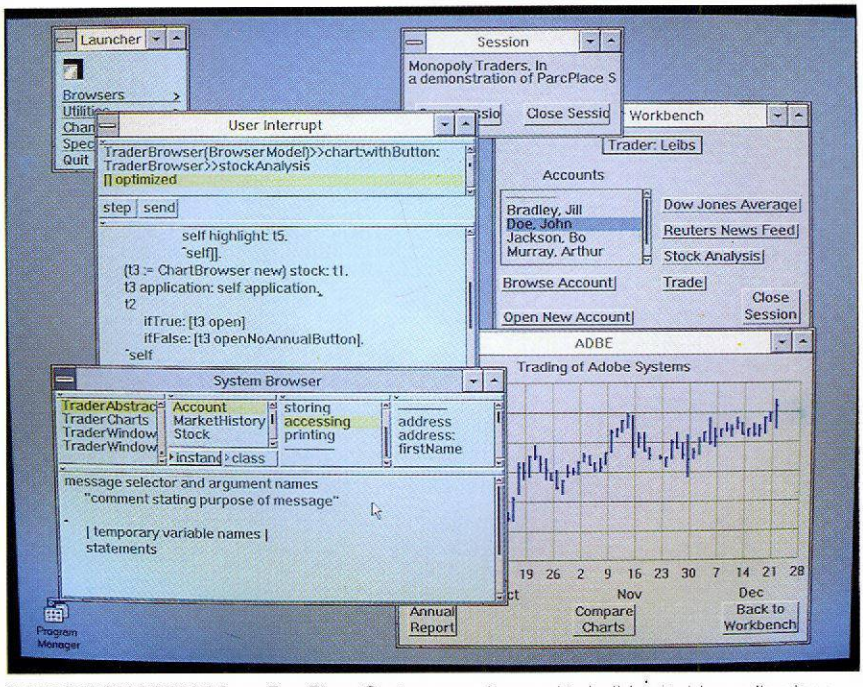
That's Smalltalk's function at Texaco Inc.'s oil exploration and land management unit in Houston. According to Texaco IS manager Dennis Samoska, the company wanted to rewrite two large mainframe applications, replacing dumb terminals with Windows-based PCs and workstations. The PCs and workstations would give users easier access to host data via advanced GUIs. After evaluating several computer-aided software engineering (CASE) tools, Samoska's unit decided to prototype and implement its new applications using ParcPlace's Objectworks\Smalltalk. Now under development, the applications will access Texaco's existing DB2 host database via an application programming interface (API).

Users like Samoska say Smalltalk takes much of the headache out of creating applications that use such GUI standards as Windows or OS/2 Presentation Manager. Rather than require programmers to learn and write to complicated GUI APIs, Smalltalk uses reusable class libraries that can link applications to the APIs. Developers can then work with an easier-to-use set of interface-building tools.

As a dynamically compiled language built on reusable objects and a virtual interface that uses machine-independent, intermediate code, Smalltalk is also easily portable between the platforms it supports. Texaco, for example, is developing its new Smalltalk applications on Sun workstations and is running them on PCs.

Help From The Big Boys

Smalltalk is also benefiting from recent recognition from IBM and Microsoft Corp. Recognizing that Smalltalk could help users write more OS/2 PM applica-



SMALLTALK\OBJECT from ParcPlace Systems can be used to build portable applications that run under several graphical interfaces.

tions, IBM endorsed the environment late last year. It signed licensing agreements with both ParcPlace and Digitalk. IBM is currently attempting to integrate Smalltalk and object-oriented programming capabilities into its AD/Cycle CASE architecture. If developers at Big Blue's Programming Systems Lab in Cary, N.C., are successful in integrating Smalltalk into AD/Cycle's information model, there's a good chance IBM will take the next step and declare Smalltalk a Systems Application Architecture (SAA)-approved language, says Cliff Reeves, manager of Common User Access for IBM.

☐ THERE ARE STILL SEVERAL MISSING CHAPTERS IN THE SMALLTALK STORY.

Not to be left behind, Microsoft is moving Smalltalk into the Windows environment. The company is working closely with ParcPlace to build low-level program-to-program messaging links into future versions of Windows and OS/2 that could help programs written in Smalltalk and other object-oriented environments such as C++ more easily share objects.

Eiffel Tower Of Babble

Public support of Smalltalk by IBM and Microsoft has given a boost to some IS managers trying to sell their bosses on the advantages of object-oriented programming and Smalltalk. "It really helps," says Phil Hartley, principal technologist in the Advanced Technology Group at American Airlines Inc.'s SABRE Computer Services. Hartley's group is currently evaluating Smalltalk along with other object-oriented and GUI-building tools such as C++, Eiffel and Object COBOL. Smalltalk currently has the strongest set of support tools, says Hartley.

Indeed, thanks to a growing number of new Smalltalk independent software vendors (ISVs), users can now choose from several sets of tools that extend Smalltalk's functionality and ease of use. Last year, Highlands, N.J.-based Synergistic Solutions Inc. started shipping its Smalltalk Platform for Integrated Computing Environments (SPICE), which includes a set of class libraries. SPICE helps link applications written in Objectworks\Smalltalk or Digitalk's Smalltalk V with the Sybase/Microsoft SQL Server and with NetBIOS or DEC-Net networking protocol interfaces. Through the Sybase database gateway, Smalltalk applications can also access DB2, Rdb and other database management systems.

Similarly, Instantiations Inc. of Portland, Ore., recently started shipping what it calls the Application Organizer Plus, a set of tools that helps large Smalltalk development teams with version management and code reuse. And at least two other small vendors, Acumen Software of Berkeley, Calif., and Tigre Object Systems Inc. of Santa Cruz, Calif., are shipping tools aimed at helping Smalltalk developers create GUI-based applications on PC, Macintosh and UNIX platforms even more easily.

Even one mainstream language vendor—Micro Focus Inc.—is getting into the Smalltalk environment. The vendor of COBOL development tools recently agreed to bundle Digitalk's Smalltalk V into its Cobol Workbench to be used to build OS/2 PM interfaces.

There are still several missing chapters in Smalltalk's story, however. Observers, note that there is still no widely accepted development methodology for Smalltalk or for any other object-oriented environment. In addition, many users are still making the transition to the relational model and structured programming techniques. "Most [IS developers] still don't know what to do with objects. They're still traumatized from making the migration to the RDBMS," says Natasha Krol, application program director at the Meta Group in Stamford, Conn. Smalltalk also faces increasingly stiff competition not only from other object-oriented languages such as C++ but also from new GUI-building tools, such as Easel from Easel Corp. and Actor from Whitewater Group.

Still, many observers see Smalltalk gaining maturity. Says Stuart Woodring, an analyst with Cambridge, Mass.-based Forrester Research Inc.: "Continued strong support from IBM and others could help push it over the top." ▭

ParcPlace Systems
1550 Plymouth Street
Mountain View CA 94043
415.691.6700
800.759.PARC
Fax 415.691.6715

Reprinted from **DATAMATION** July 15, 1991
© 1991 by CAHNERS PUBLISHING COMPANY

AUGUST 1991 SMALLTALK: 10 YEARS OF OBJECTS FEATURE

BYTE

A McGRAW-HILL PUBLICATION

Excerpted with permission, from the August 1991 issue of BYTE Magazine.
Copyright ©1991 by McGraw-Hill, Inc., New York. All rights reserved.

Smalltalk
Yesterday, Today, and Tomorrow

A look back and a look ahead at this innovative programming language—first featured 10 years ago in BYTE

L. PETER DEUTSCH AND ADELE GOLDBERG

It's been a decade since the August 1981 issue of BYTE was published. That issue provided many people with a first comprehensive look at the then-fabled Smalltalk programming environment. In this article, we look back at how people thought about Smalltalk in those days. Then we'll look more broadly at how Smalltalk and object-oriented software technology has progressed since then; we'll also consider today's state of this technology and the market for it. Finally, we'll look ahead to objects in the year 2001, another decade hence.

1981: Sending Up the Balloon

In that BYTE issue of 10 years ago, we wanted to convey three ideas about Smalltalk and object-oriented software technology: first, that an interactive, incremental approach to software development can produce qualitative and quantitative improvements in productivity; second, that software should be designed in units that are as autonomous as possible; and third, that developing software should be thought of in terms of building systems, rather than as black-box applications. The Smalltalk-80 system described in that issue so long ago was the exemplar of these three ideas.

Smalltalk was widely known then—and yet, largely unknown. Alan Kay and others from the Xerox Palo Alto Research Center (PARC) had been giving talks with tantalizing glimpses of the technology, but few people knew or understood its content. Thus, the cover of BYTE's Smalltalk issue—depicting a brightly colored Smalltalk hot-air balloon leaving an isolated island—symbolized our feeling that the time had arrived to start publicizing what we'd been doing. We believed we had new ideas that could make a real difference in how people developed software.

Many research examples developed at PARC demonstrated that object-oriented design could produce an appealing, intuitive, and direct mapping between objects in the real world and objects in a software implementation. We saw this as a radical breakthrough in one of the most difficult and problem-prone steps in software development—identifying terms and relationships as understood by human participants of a particular situation with those understood by a computer.

We believed that this simple mapping of nouns to objects was all (or most) of the story about how to design with objects, and we presented it as such in the 1981 BYTE articles. Subsequently, in examples given in our books in 1983, we demonstrated that the power of objects applied to more than nouns: It also applied to events and processes. But this power was not as well explained or exploited.

The Smalltalk research project was founded on the belief that computer technologies are the key to improving communications channels between people, in business as well as personal settings. Our activities focused on finding new ways to organize information stored in a computer and to allow more direct access and manipulation of this information.

The Smalltalk edition of BYTE introduced our approach to managing the complex information world of modern applications. It explained our methods for taking full advantage of new graphics and distributed computing and for improving the ability of experts in business and personal computing to describe their world models.

In retrospect, we are pleased that much of the software community has come to agree that the object-oriented approach to software organization is a new way to solve problems that is often better than the procedural approach. Although our ideas about problem-to-implementation mapping were incomplete—notably given the lack of format methodologies—those ideas are widely accepted today.

1991: A Decade of Experience

What have we learned in the past decade based on the Smalltalk research and experience that was introduced to the public in those 1981 BYTE articles? The first idea, as we stated earlier, is simply that a highly interactive, highly incremental software development environment can produce a qualitative improvement in software development productivity. Even in 1981, Smalltalk systems were not the only ones with this characteristic—Lisp systems pioneered the approach in the early 1960s—but they were among the outstanding examples and were the ones that moved most successfully from proprietary hardware to the microprocessor mainstream. Today, the truth of this idea is widely recognized: The suppliers of environments for more-established languages like C, C++, and Ada are now aiming to provide the benefits that Smalltalk introduced a decade ago.

BYTE ACTION SUMMARY

When BYTE first broke the news about Smalltalk to the world, there were no PC versions of the language. Now, the principles that Smalltalk pioneered have permeated the microcomputer world, and powerful versions of the language are available for a variety of personal computer platforms.

The second idea is the basic idea of object-oriented software organization: that software should be designed in units that are as autonomous as possible, should correspond to identifiable entities in the problem domain whenever possible, and should communicate through identified interfaces. This idea grows out of work on modular software design that dates back, again, to the 1960s. Object-oriented terminology adds an emphasis on direct mapping of concepts in the problem domain to software units, the idea of shared behavior and multiply instantiated state, and a focus on the interfaces between the units.

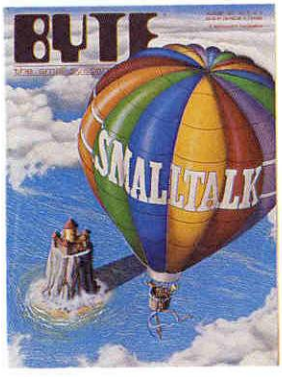
The last of these (the interfaces between the software units) makes it easy to think about systems that are configured or that grow dynamically. Smalltalk has no monopoly on new concepts, but it has been a leader in the public relations necessary to get these concepts out into the computing mainstream.

Object-oriented software organization has a natural relation to two current trends in software construction: combinable applications and open systems. Our interpretation of the term *open systems* is that for systems to grow, evolve, and combine gracefully, they should be constructed out of software with published interfaces. Functional software should be designed to be used as a component by other software, as opposed to being monolithically united with a particular interface designed only for humans at a terminal.

The third important idea that has grown partly out of the Smalltalk work is related to the open-systems idea—namely, that one should always think about building software in the context of building systems, rather than in the context of black-box applications. In other words, one should examine explicitly the nature of both the downward interfaces (the resources or facilities the software uses) and the upward interfaces (the client's use of the software) and make them as undemanding as possible. Separating functionality from the user interface, which is the Smalltalk concept of model-presentation-interaction known as *model-view-controller*, is just one application of this principle—but a very important one.

The motivation behind much of the activity in the past decade was to move Smalltalk off its island and into easy availability for the general programming community. We look at this activity as being aimed at creating a credible, concrete, and robust realization of the ideas that we could present only in sheltered research form in 1981.

As Smalltalk has moved into the commercial world, it has encountered the familiar phenomenon of technological life span. A technology comes into existence on paper, often at a university. It then progresses to research papers, research prototypes, and usable research-scale artifacts. Finally, it goes into commercial use, first by the adventurous and then by the broad mass of users—getting adapted, extended, patched, and transported as long as it continues to solve problems well, and eventually getting replaced in many or all of its uses by newer technology. Smalltalk is now in this third stage—past the scrutiny of the adventurous and experiencing wider commercial adoption.



The motivation of the past decade was to move Smalltalk off its island.

A Framework for the Future

One of the promising new concepts in object-oriented design—being actively explored today in Smalltalk as well as in other languages and environments—is the concept of a framework. In an object-oriented environment that supports inheritance, reusable software that implements a single concept frequently takes the form of a specialization hierarchy in which the superclasses are more abstract (e.g., the Smalltalk classes `Collection` and `Number`), with certain operations deliberately left to implementation by more concrete subclasses (e.g., `Array` as a concrete subclass of a kind of `Collection`, and `Integer` as a kind of `Number`). These holes in the superclasses (called *virtual functions* in C++ terminology) are an important part of the design.

A framework is a generalization of this idea to a group of classes working together. For example, the Smalltalk model-view-controller framework consists of three abstract superclasses that provide little more than definitions of how the concrete subclasses should work together, plus some bookkeeping code and default implementations of the most common operations. You reuse a framework by writing new concrete subclasses and combining existing subclasses in new ways.

Another example of a framework involves the notion of a discrete event-driven simulation, in which objects interact to represent tasks, workers, locations (where tasks are carried out by the workers), and statistically based schedules for introducing tasks and workers. New components, specialized tasks, workers, and schedules can be described in order to reuse the general framework to create specific simulations. This concept is described fully in the book *Smalltalk-80: The Language* by Adele Goldberg and Dave Robson (Addison-Wesley, 1989).

The other Smalltalk idea receiving attention today is that building software is building systems. Software should have the same property as a fractal design: Assemblies built out of parts should have the same qualitative nature (such as definable inward and outward interfaces) as those parts. Developers must realize that they cannot predict all the ways that a piece of soft-

ware will be used or all the ways that it will be ported to use the facilities of new environments.

Smalltalk in the Marketplace

One of the powerful ideas that has attracted new attention as a result of the development of object-oriented software technology is the notion of reusable, combinable applications. Today, this idea is promoted at three levels: (1) operating systems, such as Unix pipes and `fork/exec`; (2) window systems, by way of interapplication communications conventions (e.g., Apple's Interapplication Communications, Microsoft's Dynamic Data Exchange, and the X Window System's Inter-Client Communications Conventions Manual); and (3) independent software architectures (including low-level ones such as Microsoft's dynamic link libraries and Sun Microsystems' shareable libraries, as well as high-level ones such as Patriot Partners' Constellation project and ParcPlace's object model and frameworks approach).

Many believe that the discipline of defined, published interfaces—which the object-oriented approach naturally promotes—will create a new marketplace for reusable software components. However, from our experience with many developers and users of Smalltalk systems in many environments, we think the key economic shift will be in a different area.

A public market is a loosely organized environment. Components placed in a market will face a wide variety of demands, and even well-designed components with minimally constrained interfaces will have trouble attracting a critical mass of customers.

On the other hand, within a single organization, reusable components can be developed and redesigned to span a large fraction of their intended uses. In this way, the accumulation of reusable code can become an important business asset and can be treated (appropriately) as an investment and a capital good, rather than simply as a cost (which is its present treatment).

In an object-oriented environment where inheritance is supported, it is not only individual components that are reused. As we have noted, the design of interfaces between objects is often more important than the implementation of functions within objects. Frameworks can capture the structural design of software objects that address a given (partial) problem domain. As such, the frameworks developed and reused within an organization will, over time, come to capture and eventually even define the expertise of the organization—and, as such, can contribute to the organization's ability to meet its customers' needs. (This is sometimes called *competitive advantage*, but it applies equally well in situations where competition is not involved.)

2001: A Smalltalk Odyssey

If we look into our murky crystal ball, how do we see software's use of object technology in the next decade? How do we see it evolving?

We hope that in 2001, objects will be boring. In comparison, radical ideas of past decades—that system software should be written in higher-level languages or in languages with strong type systems, and that computers can and should be seamlessly networked—are thoroughly accepted today. Whether to implement them is almost never an issue now, even though there is still plenty of discussion about how to implement them well.

In the same vein, we expect that 10 years from now, the object-oriented approach to software design and implementation will be an accepted, standard technique used in every language, library, database system, and operating system and will be taught in undergraduate computer science courses at every university. This is an issue of moving the technology further out into the world, and no major new thinking will be needed to accomplish it.

One significant technological advance will be that we will free ourselves even further from equating objects with the nouns in the problem domain. Some of the most remarkable advances in the usability of computer systems have come from recognizing that processes, as well as things, can and should be described, modeled, and manipulated. Therefore, we will see software objects being used to model time, places, actions, and events. We believe that this will lead to usability advances almost as dramatic as those resulting from the now-established window/icon/mouse/pull-down interfaces that were to a large extent inspired by the original Smalltalk work of the 1970s and 1980s. ■

L. Peter Deutsch is chief scientist and Adele Goldberg is president of ParcPlace Systems (Mountain View, CA). They can be reached on BIX c/o "editors."

ParcPlace Systems
1550 Plymouth Street
Mountain View CA 94043
415.691.6700
800.759.PARC
Fax 415.691.6715

Smalltalk About Windows

Ben Smith

The Smalltalk environment has included windows since its inception. In fact, you might say that all the popular windowing environments grew out of the Smalltalk environment developed at Xerox Palo Alto Research Center (PARC). But, as with any evolving system, there are marked differences between the progenitor and its descendants.

Now, Smalltalk has recombined with the newest of the window environments, Microsoft Windows 3.0. The two major vendors of Smalltalk implementations for PCs have recently announced versions for Windows: Objectworks\Smalltalk for Windows from ParcPlace Systems, and Smalltalk/V Windows from Digitalk. While the core of both systems is Smalltalk, the Windows implementations are as different as the philosophies of the two companies.

A Question of Consistency

ParcPlace is the traditionalist; after all, the company *is* the tradition, since it spun off from the original group that developed Smalltalk at PARC. Objectworks\Smalltalk is a unique windowing environment with a mouse, window -

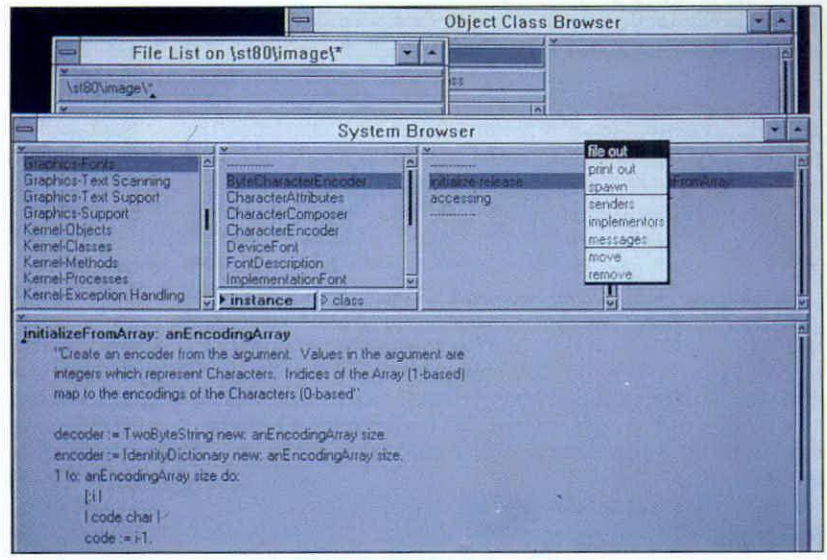
panes, scroll bars, and drop-down menus. You can use Objectworks\Smalltalk on a variety of platforms, and the window layout, icons, and window controls are always the same: the Objectworks style (see screen A).

Although this window style is not consistent with any of the newer and more widely used windowing systems, it has a great deal going for it. The most obvious feature is line wrap and rewrap: Long lines of text are wrapped around to the next line, breaking only between words. When you resize a window, the lines are rewrapped to reflect the new window size.

Another distinguishing feature of Objectworks\Smalltalk is its five-pane system browser window. (The system browser is the primary programming interface for Smalltalk.) Each pane is associated with a different function: class category, class, message category, message, and code editing. Each window pane has its own pop-up menu of operations. The pointers, icons, menus, and scroll bars maintain Objectworks' unique style on any platform.

Then there's Digitalk—the company that released the first commercial versions of Smalltalk (for DOS and then for Mac systems). Digitalk's Smalltalk/V Windows assumes that if you are programming for an established window environment, then you want to totally adopt that environment. In other words, if you develop a Smalltalk/V application for Windows, your application should look and act like a Windows application, not an application that merely runs inside of (and despite) Windows.

The drawback to this attitude is that Smalltalk/V for the Macintosh looks and acts different from the versions for Windows, plain DOS, and the X Window System. The distinct advantage of Smalltalk/V for any environment is that you can take full advantage of that environment. Your applications will be consistent with the style guidelines for that environment. For example, Smalltalk/V Windows has full access to the facilities of the Windows application programming interface, including dynamic link libraries and Dynamic Data Exchange.



Screen A: *Objectworks/Smalltalk, capital release 4 for Windows looks similar to versions of the language that run on other platforms.*

Tools and Classes

There's more to a Smalltalk implementation than a window environment and a language; there are the programming tools and the predefined class hierarchy. Smalltalk/V Windows provides fewer tools and a simpler class hierarchy than Objectworks\Smalltalk for Windows, but these limits are, in part, overcome by optional packages, like those from Digitalk and from third-party vendors such as Acumen Software. Acumen recently released a set of "user-interface construction kits" that let you develop interfaces for Smalltalk/V Mac, Smalltalk/V 286, and Smalltalk/V Windows programs—Widgets/V Mac, Widgets/V 286, and WindowBuilder/V, respectively.

Both Windows versions of Smalltalk maintain a text log of changes to the Smalltalk "image" (i.e., the Smalltalk gestalt of any moment). You can view the Smalltalk/V version of the log with the File utilities. With Objectworks\Smalltalk, you can view the change log as an object with a hierarchy that has separate instances for changes to classes, to methods, and to the system.

Both products provide a method for applying the changes of one project to another, a necessary operation if the system is to follow the objective of reusability. Both products also have an excellent debugger, as well as tools for file management, view management, and text management. As with all things, their styles differ: Objectworks maintains its own style, and Digitalk adopts the style of Windows.

Ben Smith is a technical editor for BYTE. He can be reached on BIX as "bensmith."

FORBES SUPPLEMENT ON THE INFORMATION AGE → BUSINESS AT WARP SPEED

BOSNIA AS TECH TEST: DISPATCH FROM WW3.1

ASAP Forbes

WANTED: HIGH TECH CEO. MUST BE FEARLESS, DECISIVE, FLEXIBLE, ABLE TO WORK WITH SOME VERY HEAVILY CAFFEINATED GENIUSES.

Adele

R. J. Muna

LEGENDS

Goldberg

BY NOW, HALF A CENTURY ALONG in the cyberage, most of us take for granted that when we point at an icon on our computer screen and click the mouse, what we expect to happen happens, flawlessly and almost instantaneously. That's the way of technology: Yesterday's astonishment is today's "of course." But once in a while in this everyday process, we should stop to thank Adele Goldberg.

Goldberg, 51, is one of the lesser-known but most influential members of the miracle workers at what might be called information technology's Manhattan Project, otherwise known as the Xerox Palo Alto Research Center (PARC). Back in the 1970s, those far-off days of lumbering mainframes, with Bob Taylor in the Robert Oppenheimer role, this band of revolutionaries developed much of the technology that has produced the wired world as we know it today, including the Dynabook (known in the early '70s as "the Alto"), a powerful prototype of the personal workstation, with overlapping windows in the user interfaces that ultimately led to Apple's Macintosh. While they were at it, they also developed Ethernet, laser printers, and network client servers.

Goldberg, one of the few women in the field at that time, was at PARC from 1973 to 1988 as a laboratory and research scientist. Though her name is not connected to a particular invention, such as the mouse, Goldberg's understanding of systems and the way people work with them was a key element in PARC's amazing record. PARC mentor Alan Kay describes her work at the lab as "nothing short of brilliant." During a remarkably productive 15-year period, Kay says, Goldberg designed "many of our user experiments and was central to some of the user interface development. She also took it upon herself to get Smalltalk [the breakthrough object-oriented programming language originated by Kay] out the door."

The vivacious Goldberg shrugs when asked about her place in history as one of computing's handful of women visionaries. "I'm the wrong person to ask about that," she says. "At the time, I was so involved with the work I was doing, trying to make it the best."

In 1988, spinning off her research at Xerox, Goldberg founded ParcPlace (now ParcPlace-Digitalk), currently a $50 million company that creates and sells tools for corporate application developers. She served as the company's chairman until April of this year and still holds about 2% of the company.

The air at Xerox PARC was charged with the potential sociological and philosophical effects of easier computing—the feeling that given enough power, a mouse could roar. Goldberg, coauthor of a book called *Succeeding with Objects: Decision Frameworks for Project Management*, clearly hasn't relinquished her desire to change things. Since leaving her company, she has devoted more time to projects aimed at the evolution of computer education in colleges. Computer science graduates today lack two things, says Goldberg. "One, they know programming languages but don't understand systems. Two, with some exceptions, they are used to working alone. They don't know how to work in teams." Currently, she is helping create new computing science courses at community colleges in the United States and at universities abroad. She has also been involved, as a board member and mentor, in Cogito Learning Media, a new company formed to provide multimedia software for science education.

At ease in her sunny, commodious Palo Alto, California, home, Goldberg gestures excitedly when she starts talking about education. This is where it all began for her. With a doctorate in information science from the University of Chicago, Goldberg was doing research in education technology at Stanford when Kay recruited her to head the pedagogic group at Xerox PARC. Initially, the Dynabook efforts were applied to educational uses, which, happily for post-1984 computer tyros, put a premium on ease of use.

During her years at PARC, Goldberg worked in local Palo Alto schools, bringing students into the lab to experiment with the Alto. Kay remembers once in the mid-1970s when she wanted to take the then-revolutionary hardware to a nearby middle school for on-site learning experiments. "After going through all the work of making the first modern personal computer for children," Kay says, "Xerox balked." Finally, Kay and Goldberg pulled Goldberg's station wagon up to the research center's front door, loaded the machines, and took them down to the school. There were no repercussions, except for the enthusiasm of the teachers and kids, and the machines stayed in the school for a full year.

This was a pivotal moment in Goldberg's education, too, teaching her the valuable lesson that it's better to say you're sorry than to ask permission. The incident, says Goldberg, "helped start my career as a troublemaker." —*Umberto Tosi*

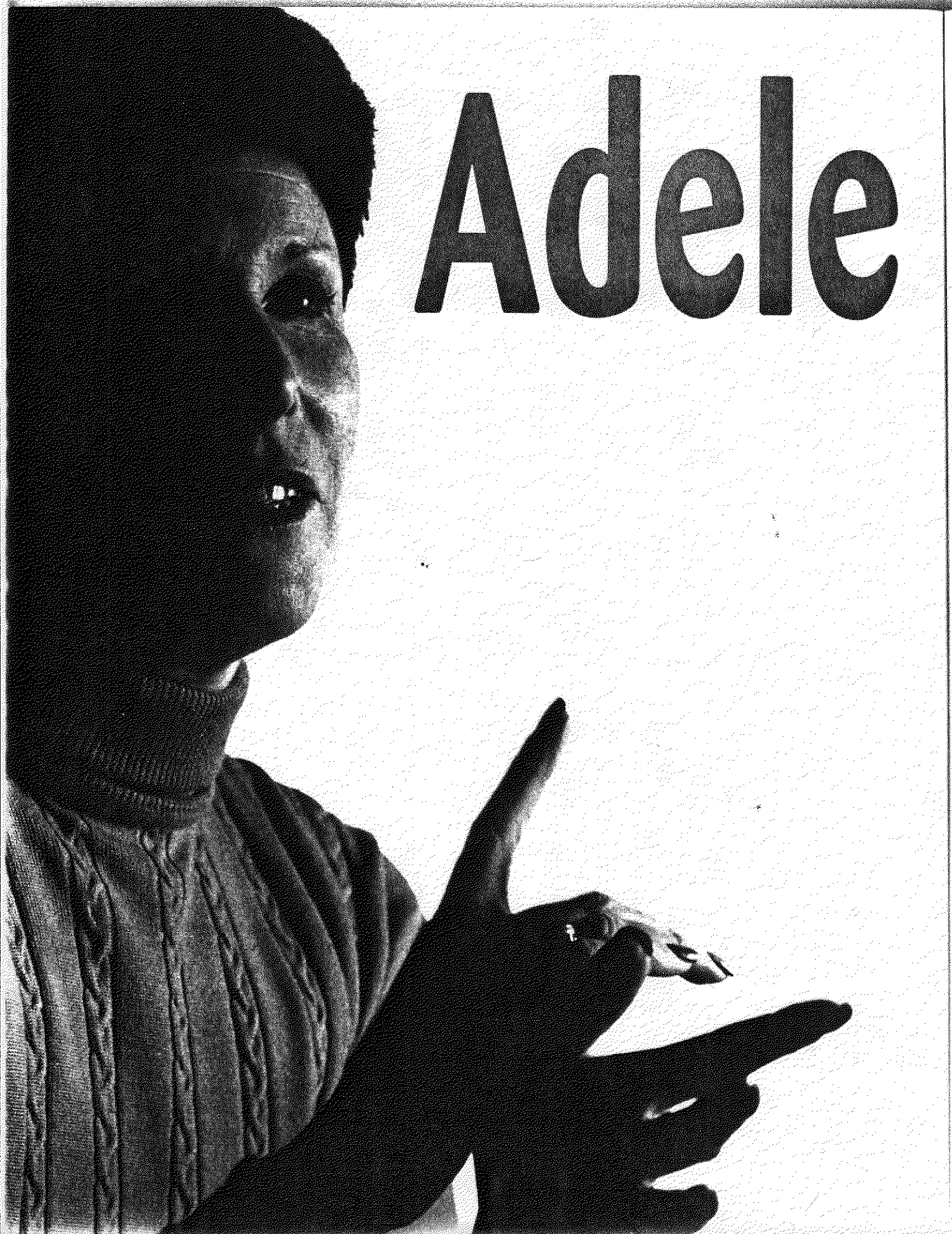
R. J. Muna

BOSNIA AS TECH TEST: DISPATCH FROM WW3.1

Forbes

WANTED: HIGH TECH CEO. MUST BE FEARLESS, DECISIVE, FLEXIBLE, ABLE TO WORK WITH SOME VERY HEAVILY CAFFEINATED GENIUSES.

PLUS: SURVIVAL TOOLS FOR THE INFORMATION SUPERHIGHWAY



R. J. Muna

LEGENDS

Goldberg

BY NOW, HALF A CENTURY ALONG in the cyberage, most of us take for granted that when we point at an icon on our computer screen and click the mouse, what we expect to happen happens, flawlessly and almost instantaneously. That's the way of technology: Yesterday's astonishment is today's "of course." But once in a while in this everyday process, we should stop to thank Adele Goldberg.

Goldberg, 51, is one of the lesser-known but most influential members of the miracle workers at what might be called information technology's Manhattan Project, otherwise known as the Xerox Palo Alto Research Center (PARC). Back in the 1970s, those far-off days of lumbering mainframes, with Bob Taylor in the Robert Oppenheimer role, this band of revolutionaries developed much of the technology that has produced the wired world as we know it today, including the Dynabook (known in the early '70s as "the Alto"), a powerful prototype of the personal workstation, with overlapping windows in the user interfaces that ultimately led to Apple's Macintosh. While they were at it, they also developed Ethernet, laser printers, and network client servers.

Goldberg, one of the few women in the field at that time, was at PARC from 1973 to 1988 as a laboratory and research scientist. Though her name is not connected to a particular invention, such as the mouse, Goldberg's understanding of systems and the way people work with them was a key element in PARC's amazing record. PARC mentor Alan Kay describes her work at the lab as "nothing short of brilliant." During a remarkably productive 15-year period, Kay says, Goldberg designed "many of our user experiments and was central to some of the user interface development. She also took it upon herself to get Smalltalk [the breakthrough object-oriented programming language originated by Kay] out the door."

The vivacious Goldberg shrugs when asked about her place in history as one of computing's handful of women visionaries. "I'm the wrong person to ask about that," she says. "At the time, I was so involved with the work I was doing, trying to make it the best."

In 1988, spinning off her research at Xerox, Goldberg founded ParcPlace (now ParcPlace-Digitalk), currently a $50 million company that creates and sells tools for corporate application developers. She served as the company's chairman until April of this year and still holds about 2% of the company.

The air at Xerox PARC was charged with the potential sociological and philosophical effects of easier computing—the feeling that given enough power, a mouse could roar. Goldberg, coauthor of a book called *Succeeding with Objects: Decision Frameworks for Project Management*, clearly hasn't relinquished her desire to change things. Since leaving her company, she has devoted more time to projects aimed at the evolution of computer education in colleges. Computer science graduates today lack two things, says Goldberg. "One, they know programming languages but don't understand systems. Two, with some exceptions, they are used to working alone. They don't know how to work in teams." Currently, she is helping create new computing science courses at community colleges in the United States and at universities abroad. She has also been involved, as a board member and mentor, in Cogito Learning Media, a new company formed to provide multimedia software for science education.

At ease in her sunny, commodious Palo Alto, California, home, Goldberg gestures excitedly when she starts talking about education. This is where it all began for her. With a doctorate in information science from the University of Chicago, Goldberg was doing research in education technology at Stanford when Kay recruited her to head the pedagogic group at Xerox PARC. Initially, the Dynabook efforts were applied to educational uses, which, happily for post-1984 computer tyros, put a premium on ease of use.

During her years at PARC, Goldberg worked in local Palo Alto schools, bringing students into the lab to experiment with the Alto. Kay remembers once in the mid-1970s when she wanted to take the then-revolutionary hardware to a nearby middle school for on-site learning experiments. "After going through all the work of making the first modern personal computer for children," Kay says, "Xerox balked." Finally, Kay and Goldberg pulled Goldberg's station wagon up to the research center's front door, loaded the machines, and took them down to the school. There were no repercussions, except for the enthusiasm of the teachers and kids, and the machines stayed in the school for a full year.

This was a pivotal moment in Goldberg's education, too, teaching her the valuable lesson that it's better to say you're sorry than to ask permission. The incident, says Goldberg, "helped start my career as a troublemaker." —*Umberto Tosi*

R. J. Muna

Course Number	CSE 503
Course Title	Software Process Practicum: Lessons in Software Quality and Leadership
Instructors	Judy Bamberger and James Hook
Days	Mondays, Wednesdays (and one Saturday)
Times	5:00pm - 7:00pm
Room	Cooley Center (CC) 371
Number of Units	4 credits

The software process practicum is designed to immerse the working student in topics relevant to software process improvement and quality management, and to introduce them to the supporting theory. Topics include process management frameworks (capability maturity model, ISO 9000), measurement for process improvement, and key team skills necessary for effective collaborative software engineering efforts. At the end of the course the student will be able to demonstrate that the software development process can be managed and controlled, leading to increased software quality. In addition to lectures and in-class "labs," the class will include one Saturday workshop.

OBJECTIVES / VISION

After this class, you, the students will understand and have demonstrated that:

- Software processes can be managed and controlled.
- Software engineering is a social process, too.
- You have real skills that you can apply today at work.
- You have a framework on which to build your own educated decisions about applying software quality principles and tools to personal, project, and corporate software activities.
- You have identified three things to improve at your own work place (or within your own personal process), and you have begun working on them.

© 1996 Judy Bamberger / James Hook

To the Students:

This is a list of required and recommended books.

Readings will be derived from the three required books throughout the semester. We will be discussing some of them as part of the class session. They present a unique view of many of the concepts, models, and skills we will be covering - often a different view than would be found in most computer science courses.

The recommended books will provide additional breadth and assistance throughout the class.

We have selected these books because we believe they will be useful to you after this course, in your work environment and in your professional activities. Your feedback throughout the course and afterward will be appreciated.

Required Books

(1) Grady, Robert B and Caswell, Deborah L, *Software Metrics: Establishing a Company-Wid Program*, P T R Prentice Hall, 1986

(2) Scholtes, Peter R et al, *The TEAM Handbook*, Joiner, 1998

(3) Weinberg, Gerald M, *Quality Software Management, Volume 1, Systems Thinking*, Dorset House Publishing, 1993

Recommended Books

(1) Brassard, Michael, *The Memory Jogger Plus+*, GOAL/QPC, 1989

(2) Weinberg, Gerald M, *Quality Software Management, Volume 2, First-Order Measurement*, Dorset House Publishing, 1993

© 1996 Judy Bamberger / James Hook

ORGANIZING FOR SUCCESS

Things to think about from the beginning:

- Who would you like on your team? Teams are not required to keep the same members throughout the entire course. However, especially once the mid-term project is begun, this does have significant advantages. We suggest you begin now, and think about how you could build an excellent, effective, and high-performing team (we will be giving you some hints, too). We also suggest that you use your first team project to try some of those ideas, and set the tone for success.
- With which "partner organization" would you like to work? The mid-term and end-term assignments will focus on working with what we call a "partner organization." This could be a team with which you work at your company or school (highly preferred), or it could be a team we recommend to you. You will be collaboring with them to define a process and then to create a plan to improve that process. This will involve some of their time - in the past, it has been about 2 - 12 hours total over the entire term (depends on number of people involved, and depth of their involvement).

© 1996 Judy Bamberger / James Hook

GRADING CRITERIA

There will be individual and team assignments. We have tried to give most assignments on a Monday, with the turn-in date generally on the following Monday. Team assignments will be given a single grade, which will be assigned to each team member. Team assignments will have an individual component associated with each (graded individually) to analyze team effectiveness overall, and the individual's effectiveness within that team.

The goal of these assignments is to allow you to reinforce the concepts and skills learned in one or more Practicum sessions.

There will be a mid-term and end-term project to be done as a team. The mid-term and end-term projects are related (general descriptions are included in the syllabus). The mid-term project focuses on working with a partner organization to define a software-related process using the techniques we learn in Practicum. The end-term project focuses on working with that partner organization to identify and plan for improvements to that process.

The goal of these projects is to allow you to synthesize the concepts and skills learned in several Practicum sessions and practice them in a real-world setting. Past projects have also resulted in significant benefits to the partner organization as well, a secondary goal.

There will be a final project to be done individually. This project will be to create an improvement plan for a process in which you are involved personally - individually or as part a team at work or outside of work.

The goal of this project is to allow you to synthesize the information learned in Practicum and apply it in a real-world, relevant context.

There are two un-graded elements as well.

The goal of both of these are to help us continuously improve the Practicum - both for you, this term, and for future offerings of the Practicum.

We will be asking you to keep a Timelog - the amount of time you spend preparing for each class (e.g., reading) and doing the homework assignments. This will have absolutely no bearing on any grade. In fact, we will not look at it until any related assignments have been graded. We will use this to help us assess and tune the overall workload, week-by-week, assignment-by-assignment. A Timelog template (with instructions) is included in this syllabus.

Please turn in your Timelog sheets each class session.

We will also be asking you to keep a Journal - short notes about the readings and learnings. We will ask to see this three times during the term. We will use this to help us assess the impact of the readings and the messages you take from the classes. Again, this will have absolutely no bearing on any grade, and we will not look at it until related work has been graded. We will use this to help us identify "what works" and "what doesn't," as well as those articles, class sessions, exercises, etc that have the most/least impact. A set of suggested items to cover in your Journal entries is included in this syllabus.

Please try to make your Journal entries each day, or as new learnings come to you.

© 1996 Judy Bamberger / James Hook

• Homework - individual/team	20%
• Mid-term project - team Process Definition	20%
• End-term - team Process Improvement	30%
• Final project	30%
• Class participation (individual)	subjective
• Timelog	0%
• Journal	0%

We will make every effort to return homework to you within one calendar week. You will see comments from us and the following notations in the upper right corner, with the following meanings:

Φ	"not"; no grade noted; perhaps our instructions were not clear; goals of the assignment were missed; please see us and let's get straightened out; OK to rework and resubmit for success
–	"minus"; does not meet minimal criteria; OK to rework and resubmit for success
√	"check"; meets goals of homework assignment
+	"plus"; exceeds goals of homework assignment
++	"double-plus"; exceeds our wildest dreams !!!

OFFICE HOURS and CONTACT INFORMATION

- Judy Bamberger: Monday, 7:00pm - 8:00pm at OGI and by appointment
- Jim Hook: Wednesday, 7:00pm - 8:00pm at OGI and by appointment

Judy Bamberger	Jim Hook
690-1206	690-1169
bamberg@cse.ogi.edu	hook@cse.ogi.edu
room Strawberry office	room CSE 143

We are here to ensure you get the most out of this class, so please come and talk to us when you need to! If you use Email to communicate or ask questions, then please send it to both of us.

© 1996 Judy Bamberger / James Hook

BIBLIOGRAPHY AND SYLLABUS

In the following:

(R#) means there is reading to be done before this class

In general, the reading is intended to be done to a level where you are confident you can discuss the key themes (maybe not all the details), as we will begin many of the classes with a discussion of the readings. The goal of assigning the readings is to broaden your background, and to provide you with "intellectual pointers" to key references for the class (short term) and for your future as a professional (longer term).

Those few times where detailed understanding of the readings is required (e.g., to prepare for a specific class discussion or a homework assignment), we will indicate that explicitly. Whenever in doubt, one way or another, please ask.

(H#) means there is homework to prepare after this class

*(H#) The asterisk (*) before the homework indicates there is an individually-done assignment to be turned in

&(H#) The ampersand (&) before the homework indicates there is a team-done assignment to be turned in (i.e., one single assignment per team, with all team member names on it)

(H#) The lack of any leading marking before the homework indicates this is reading, other material, or other activities related to completing some other homework assignment to be turned in

(JOUR) means there is a journal-related activity here

The • -ed and – -ed lists summarize the topics to be covered in this class

When you turn in your homework, please make sure the following are done:

- Please put your name clearly on the front page, and indicate which homework this is (at least the H-number you see in the syllabus and on any relevant handouts)
- Please put page numbers on each page (hand-written is OK)
- Please type (vs hand-write) your assignments (with many papers to read, we find it difficult and slow with a lot of hand-written papers)
  - Do not spend a lot of time on formatting; just leave us enough space to write some comments and ask some questions
- Please run some sort of spell-checker on your papers and make corrections (some of the typographical errors significantly decrease our ability to "figure out" what you are trying to say)

© 1996 Judy Bamberger / James Hook

Handouts, Articles, Readings, and Other Materials Passed to Students

(1) Introduction, Ice Breaking, and Motivation

Week: 1	Class: 1	Day/Date: Monday, 30 September	Key Presenter: Jim and Judy	Readings? no Homework? yes

- Copy of class materials
  - Develop common expectations about the class
  - Communicate class mechanics
  - Briefly survey "quality"
  - Discuss quality in the software context by introducing an example

*(H1a) Write your "process biography" (per instructor-provided questionnaire, handed out in class)
[turn in at class #2]

*(H1b) Interview three software development organizations (per instructor-provided questionnaire, handed out in class)
[turn in at class #3]

(JOUR) Write a "learning contract" for yourself - what do you want to learn; how do you want to learn it; how, when, where, with whom do you want to practice it; how will you claim "success" for yourself after Practicum is over; make this your first Journal entry

(2) Background: Statistical Process Control; Several Basic Quality Tools

Week: 1	Class: 2	Day/Date: Wednesday, 2 October	Key Presenter: Jim and Judy	Readings? yes Homework? yes (thinking only)

(R2a) *Deming Management at Work;* Mary Walton; Chapter 2, "Florida Power & Light"

(R2b) *Quality Planning and Analysis - From Product Development through Use;* J M Juran and Frank M Gryna; Chapter 1, "Basic Concepts" and Chapter 2, "Companywide Assessment of Quality"

(R2c) *Quality is Free;* Philip B Crosby; Chapter 2, "Quality May Not Be What You Think It Is," Chapter 3, "The Quality Management Maturity Grid," and the Browser's Guide

© 1996 Judy Bamberger / James Hook

(R2txt) *Quality Software Management, Volume 1, Systems Thinking;* Gerald M Weinberg; Chapter 1, "What Is Quality? Why Is It Important?"

- Copy of class materials
  - Introduce several quality tools (brainstorming, consensus, affinity diagram, Pareto diagram)
  - Discuss classical Statistical Process Control (SPC)
  - Point to other quality and management and planning tools (flowchart, check sheet, run chart, historgram, scatter diagram; interaction digraph, tree diagram, prioritization matrices, matrix diagram, process decision program chart, activity network diagram)

(H2a) Begin thinking about how you will build your team for class projects

(3) How does the "I" fit into "TEAM"?

Week: 2	Class: 3	Day/Date: Monday, 7 October	Key Presenter: Judy	Readings? yes Homework? yes

(R3a) *Social Style Profile - Feedback Booklet;* Wilson Learning

(R3b) *Enterprise,* Winter 1991/92; "Unleashing People Power - Innovation comes from the Individual at Chaparral Steel"

(R3c) *Training & Development Journal,* April 1991; Richard Wellins and Jill George; "The Key to Self-Directed Teams"

(R3txt) *The Team Handbook,* Chapter 4, "Getting Underway"

[optional] *The Team Handbook,* Chapter 6, "Learning to Work Together"

- Copy of class materials
  - Meeting management techniques
  - Different style preferences
  - Team development and growth
- Handout - "Teams Need a Common Goal" (Hagar the Horrible cartoon)
- Handout - "Why Some Teams Don't Fail" (from *Manage,* July 1993)

*(H3a) Analyze the strengths and weaknesses of each of the social styles (per instructor-provided scenario, handed out in class)
[turn in at class #5]

© 1996 Judy Bamberger / James Hook

(H3b) [readings to be handed out in class] *Please Understand Me - Character & Temperament Types*; David Keirsey and Marilyn Bates; Chapter 1, "Different Drums and Different Drummers" and Appendix: The Sixteen Types

(H3c) [reading to be handed out in class] "MBTI Short Summary", Judy Bamberger

*(H3d) Do the readings above; see what the instrument indicates as natural tendencies, and discuss (3-5 pages) how these characteristics manifest themselves in your team interactions at work
[turn in at class #5]

(4) Problem Solving Paradigms

Week: 2	Class: 4	Day/Date: Wednesday, 9 October	Key Presenter: Judy	Readings? yes Homework? yes

(R4a) [optional] *Problem-Solving Process*; Xerox, 1992; "Participant Guide"

(R4txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 2, "Software Subcultures" and Chapter 3, "What is Needed to Change Patterns?"

- Copy of class materials
  - One problem solving model (useful tools at each phase, decision making styles and tools)
  - One conflict resolution model (identifying, managing, and resolving conflict)
- Handout - Role Play: Veginots (if used in class)

&(H4a) As a team, using the basic tools, discuss how you would solve the problem of whether or not to inform the customer of a potential schedule slippage (per instructor-provided guidelines, handed out in class)
[turn in at class #6]

(5) Process Definition Techniques

Week: 3	Class: 5	Day/Date: Monday, 14 October	Key Presenter: Judy	Readings? yes Homework? optional

(R5a) *Managing the Software Process*; Watts Humphrey; Preface, Part One, "Software Process Maturity", Chapter 2, "The Principles of Software Process Change", Chapter 14, "The Software Engineering Process Group", Chapter 20, "Conclusion"

© 1996 Judy Bamberger / James Hook

(R5b) *IBM Systems Journal*, 1985; R A Radice *et al*; "A Programming Process Architecture"

- Copy of class materials
  - Several techniques for representing defined processes
- Handout - Process Definition Examples
- Handout - "Rules of thumb for developing processes - Lessons learned from the trenches"; Mary Sakry

*(H5a) [optional] Demonstrate the process representation and definition techniques on the sample process description we provide (handed out in class)
[turn in at class #7]

Mid-Term Homework

- Define a process with your partner organization (per instructor-provided guidelines; draft attached, handed out in class)
  [turn in at class #10; make presentation at class #11]

(6) Continuation of Topic from class #5

Week: 3	Class: 6	Day/Date: Wednesday, 16 October	Key Presenter: Judy	Readings? yes Homework? nothing new

Please do the following readings very carefully. The key idea to learn is the method Weinberg uses to represent and analyze processes (the drawings with "blobs" and annotated arrows). We will be using this in class and/or as part of a future assignment.

(R6txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 4, "Control Patterns for Management" and Chapter 5, "Making Explicit Management Models"

(7) Continuation of Topic from class #5

Week: 4	Class: 7	Day/Date: Monday, 21 October	Key Presenter: Judy	Readings? yes Homework? nothing new

(JOUR) Please turn in your Journals.

Please do the following readings very carefully. The key idea to learn is the method Weinberg uses to represent and analyze processes (the drawings with "blobs" and annotated arrows). We will be using this in class and/or as part of a future assignment.

© 1996 Judy Bamberger / James Hook

(R7txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 6, "Feedback Effects," Chapter 7, "Steering Software," and Chapter 8, "Failing to Steer"

- Note: If we complete planned in-class exercises early, we may do a student-provided activity or an exercise based on the Weinberg reading, or we may start the next topic
- ANNOUNCING MAJOR READING FOR class #9!!!

  Select two quality models (in class) and be prepared to present and discuss a set of comparison issues (per instructor-provided criteria); this will involve careful reading (vs detailed skimming)

(8) Capability Maturity Model for Software (CMM)

Week: 4	**Class: 8**	**Day/Date:** **Wednesday, 23 October**	**Key Presenter: Judy (Jim is out)**	**Readings? yes** **Homework? nothing new**

(R8a) *Capability Maturity Model for Software, Version 1.1*; Software Engineering Institute; Chapter 1, "The Process Maturity Framework", Chapter 3, "Operational Definition of the Capability Maturity Model", Appendix A, "Goals for Each Key Process Area"

(R8b) *IBM Systems Journal*, 1985; W S Humphrey; "The IBM large-systems software development process: Objectives and direction" and R A Radice *et al*; "A programming process study"

(R8c) *American Programmer*, September 1994; James Bach, "The Immaturity of the CMM"

--- and optional readings (helpful references) ---

(R8d) *IEEE Software*, July 1994; Michael K Daskalantonakis; "Achieving Higher SEI Levels"

- Copy of class materials
  - Basic process management concepts
  - How they apply to software engineering
  - Characterization of "immature" and "mature" software engineering processes
  - Five levels of maturity as defined by CMM
  - Details of Repeatable Level
  - Framework of CMM - applicability across many disciplines
  - Some anecdotal data on ROI

© 1996 Judy Bamberger / James Hook

(9) Quality Frameworks: Applying the Concepts to Process Improvement

Week: 5	Class: 9	Day/Date: Monday, 28 October	Key Presenter: Judy	Readings? yes Homework? nothing new

(R9a) *International Standard, ISO 9000-3*; ISO; "Quality Management and Quality Assurance Standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software"

(R9b) *Trillium*, 1994; Bell Canada; "Telecom Software Product Development Process Capability Assessment"

(R9c) *Quality System Review - Guidelines*, March 1995; Motorola Corporate; "Introduction", "QSR General Scoring Maturity Matrix", "Subsystem 10 - Software Quality Assurance" and "Scoring Reference", "Quality Policy for Software Development"

(R9d) *Malcolm Baldrige National Quality Award*, 1996

- Completion of previous material and compare/contrast all quality models

(10) Quality Technique #1 - Formal Inspections

Week: 5	Class: 10	Day/Date: Wednesday, 30 October	Key Presenter: Judy	Readings? yes Homework? yes

**** It's the day before Halloween; dress up; party!!! ****

(R10a) Neal Brenner; "The ST Inspection Handbook"

(R10b) *IBM Systems Journal*, 1976; M E Fagan; "Design and code inspections to reduce errors in program development"

(R10c) *IEEE Software*, July 1994; Robert B Grady and Tom Van Slack; "Key Lessons in Achieving Widespread Inspection Use"

(R10d) *IEEE Software*, March 1994; Jack Barnard and Art Price; "Managing Code Inspection Information"

--- and optional readings (helpful references) ---

(R10e) *IEEE Software*, September 1993; Edward F Weller; "Lessons from Three Years of Inspection Data"

(R10f) *IEEE Transactions on Software Engineering*, July 1986; Michael E Fagan; "Advances in Software Inspections"

© 1996 Judy Bamberger / James Hook

(R10g) *Software Validation - Inspection - testing - verification - alternatives*; A F Ackerman, P J Fowler, and R G Ebenau; "Software Inspections and the Industrial Production of Software"

(R10h) *Software Inspection*; Tom Gilb and Dorothy Graham; "Software Inspections at Applicon" by Barbara Spencer

- Copy of class materials
  - One defined process for formal inspections
  - One set of metrics that can be obtained from formal inspections
  - One set of forms, guidelines, rule sheets for formal inspections
- Handout - Inspection Package
- Other handouts to support the lab will be provided as needed

(H10a) Prepare for formal inspection workshop, Saturday (use inspection lab materials, handed out in class)
[class #11]

(11) Inspection Workshop and presentation of Mid-Term Homework: Process Definition

Week: 5	Class: 11	Day/Date: Saturday, 2 November	Key Presenter: Judy and students	Readings? nothing new Homework? nothing new

End-Term Homework

- With your partner organization, define a process improvement plan for the process you defined in the Mid-Term Homework (per instructor-provided guidelines; draft attached, handed out in class)
[turn in at class #16; presentation at class #17]

(12) Process Improvement Models

Week: 6	Class: 12	Day/Date: Monday, 4 November	Key Presenter: Judy	Readings? yes Homework? optional

(R12a) selections from the *Quality* issue of Business Week, 1991

(R12b) [optional] *Total Quality Improvement System*; ODI; "Quality Action Teams - Team Member's Workbook"

© 1996 Judy Bamberger / James Hook

(R12c) [optional] *Total Quality Improvement System*; ODI; "Quality Action Teams - Project Booklet"

(R12txt) *The Team Handbook*, Chapter 5, "Building an Improvement Plan"

- Copy of class materials
  - Several process improvement models and strengths of each
  - Principles of action planning and "how to"
- Handout - Organizational Climate Survey, Judy Bamberger

*(H12a) [optional] Leveraging the readings from Business Week and comparing it with your experience, discuss what appears to you to be the "top six" characteristics of high-quality organizations
[turn in at class #14]

(13) Software Metrics

Week: 6	Class: 13	Day/Date: Wednesday, 6 November	Key Presenter: Judy	Readings? yes Homework? yes

(R13a) *Software Modeling and Measurement: The Goal/ Question/Metric Paradigm*, Victor R Basili

(R13b) [optional] *Software Quality*, "Software Metrics that Meet your Information Needs," Linda Westfall

(R13txt) *Software Metrics: Establishing a Company-Wide Program*, by Robert B Grady and Deborah L Caswell, chapters 5-6 and 12-15 (please use this as a minimum guideline; we would have like to have assigned the entire book)

- Copy of class materials
  - Why measure software processes
  - What are some things that can be measured in software processes
  - Goal, Question, Metric Paradigm
  - Examples

*(H13a) Identify a problem at work and do a "detailed impact case study" and a "subjective impact study" following Weinberg, Volume 2, sections 8 (especially 8.4 - 8.6) and 9 (especially 9.3 - 9.5) (handed out in class)
[turn in at class #15]

© 1996 Judy Bamberger / James Hook

Week: 7	**NO CLASS**	**Day/Date:** **Monday,** **11 November**		**Readings? nothing new** **Homework? nothing new**

Week: 7	**NO CLASS**	**Day/Date:** **Wednesday,** **13 November**		**Readings? nothing new** **Homework? nothing new**

(14) Organizational Infrastructure for Sustained Process Improvement - Leadership and Technology Transition

Week: 8	**Class: 14**	**Day/Date:** **Monday,** **18 November**	**Key Presenter: Judy**	**Readings? yes** **Homework? nothing new**

(JOUR) *Please turn in your Journals.*

(R14a) *Quality Planning and Analysis - From Product Development through Use*; J M Juran and Frank M Gryna; Chapter 7, "Organization for Quality" and Chapter 8, "Developing a Quality Culture"

(R14b) *The Leadership Challenge*; James Kouzes and Barry Posner; Part One, "Knowing What Leadership Is Really About", Chapter 1, "When Leaders Are at Their Best: Five Practices and Ten Commitments", Chapter 2, "What Followers Expect of Their Leaders: Knowing the Other Half of the Story", Part 7, "The Beginning of Leadership", Chapter 13, "Become a Leader Who Cares and Makes a Difference"

(R14c) "Leading Change: Why Transformation Efforts Fail"; *Harvard Business Review*; March/April 1995

(R14txt) *Quality Software Management, Volume 1, Systems Thinking*; Gerald M Weinberg; Chapter 18, "What We've Managed to Accomplish"

- Copy of class materials
  - Examined some management/leadership issues to sustain process improvement
  - Discussed key points of effective leadership
  - Understand your role as leader for process improvement
  - Getting information out about improved process
  - Getting improved process adopted and used

© 1996 Judy Bamberger / James Hook

(15) Quality Technique #2 - Quality Function Deployment (QFD)

Week: 8	Class: 15	Day/Date: Wednesday, 20 November	Key Presenter: Judy	Readings? yes Homework? nothing new

(R15a) *Harvard Business Review*, May-June 1988; John Hauser and Don Clausing; "The House of Quality"

(R15b) "QFD for Software - Satisfying Customers"; Richard Zultner

- Copy of class materials
  - Increasing importance of focus on quality
  - Voice of the customer
  - Exercise using QFD
- Handout - Leemak, Inc; "Problems That QFD Solves"
- Handout - Leemak, Inc; "OK, So How Long Does It Really Take?"

(16) Managing Change

Week: 9	Class: 16	Day/Date: Monday, 25 November	Key Presenter: Judy	Readings? yes Homework? nothing new

(R16a) *IEEE Software*, January 1990; Barbara M Bouldin, "The nature of change agents"

(R16b) *Harvard Business Review*, January-February 1992; Robert H Schaffer and Harvey A Thomson, "Successful Change Programs Begin with Results"

(R16c) *Group and Organization Studies*, SAGE Publiciations, Group and Organizational Studies, December 1982; J Scott Armstrong, "Strategies for Implementing Change: An Experiential Approach" (Delta process)

- Copy of class materials
  - People issues about key principles of effective change

(17) Team Presents: End-Term Homework

Week: 9	Class: 17	Day/Date: Wednesday, 27 November	Key Presenter: students	Readings? yes Homework? nothing new

© 1996 Judy Bamberger / James Hook

(R17a) *Teaching the Elephant to Dance*; James Belasco; Chapter 1, "Teaching the Elephant to Dance - The Manager's Guide to Empowering Change", Chapter 2, "Getting Ready to Change", Chapter 11, "Empower Individual Change Agents", Chapter 12, "Change Happens - The Elephant Learns"

(R17b) Article from Wall Street Journal, 13 September 1994 on Chinese Quality Managers

(18) Surprise !!!

Week: 10	**Class: 18**	**Day/Date:** **Monday,** **2 December**	**Key Presenter: Jim, entire class (Judy is out)**	**Readings? nothing new** **Homework? nothing new**

(19) Review and Summary

Week: 10	**Class: 19**	**Day/Date:** **Monday,** **4 December**	**Key Presenter: discussion (Judy is out)**	**Readings? nothing new** **Homework? nothing new**

(JOUR) Please turn in your Journals.

Final Project

- As an individual, define a process improvement plan for a personal, project, or organizational process in which you have a stake (per instructor-provided guidelines, handed out in class)
  [turn in to Jim Hook or Judy Bamberger at OGI no later than Thursday, 12 December 1996]

© 1996 Judy Bamberger / James Hook

Journal Entry Ideas

To the Students:

The readings we suggest are intended to meet the goal of providing growth and broadening, rather than something we will "test" in class. To help us evaluate the effectiveness of the readings in meeting this goal, we would like you to use a "journal" to capture your reactions to the readings.

This is also a perfect opportunity to capture some of your thoughts about the class sessions themselves. While we do a course evaluation at the end of the term, and while we ask for input at various places throughout the term, we have heard from our students that they would find it easier to capture evaluation thoughts as the class progresses. You can use that information when it is time to do the final course evaluation.

This is your journal to keep - any way you want - hand scribbled notes and diagrams, typed and indexed on a computer - whatever. We will ask to see it three times throughout the course (approximately week 4, week 7, and week 10) so we can assess where we are going with the course. If you would rather we did not see your Journal, as it may contain private insights, we request to meet briefly with you to assess the impact of the readings and class sessions.

No grades will be given. No comments will be made, unless you explicitly request us to do so. We will use your input to make mid-course corrections and improvements in the Practicum.

Some topics you might want to consider:

- How does a reading, homework, or class session help you meet your learning contract?
- What key points did you get out of the reading, homework, or class?
- How does this synthesize with previous readings, homeworks, and classes - what new "ahas" came?
- What new knowledge did you gain from the reading, homework, or class?
- What new puzzles are opening up for you - areas where you want to experiment or get more knowledge?
- ... anything that strikes you as important at the time you write.

© 1996 Judy Bamberger / James Hook

TIMELOG TEMPLATE

To the Students:

We would like to collect information on the time each student spent on each class preparation and assignment. We will use this ***ONLY*** to help us evaluate and tailor the workload we are asking from each of you. The preferred format to give us time spent is:

hours : minutes

with a granularity of 15 minutes.

We will ***NOT*** use time spent as a marking criterion in any way. It will be maintained in a separate database, and not examined until after all marks are given for that assignment.

Our goal is to have an "achievable class" - with some planning on your part and our part. Your providing us with this information can help us determine if we are meeting that goal. Thank you.

© 1996 Judy Bamberger / James Hook

General Information

Student Name	
Class #	

Readings

Reading #s					
Time spent					

Homework

Homework #s					
Time spent					

Projects (Mid-Term, End-Term, Final)

Project ID					
Time spent					

Other

Description					
Time spent					

Complete only if turning in homework today

Homework ID					
Due Date					
Turn-In Date					

© 1996 Judy Bamberger / James Hook

Mid-Term Project: Process Definition

To the Students:

This exercise is to be done as a team. Once again, "team" is defined as about three people. Effective meeting management and team collaboration skills are key to success of this exercise.

This exercise builds on classes #5, #6, and #7 (process definition) and additional readings and information about process definition. It is due, in writing, at class #10, Wednesday, 30 October 1996. Each team will present its results at class #11, Saturday, 2 November 1996, in the afternoon.

The assignment is to produce a process definition, using the techniques and representations we will be sharing with you this week, or others you may have used or know. You are not constrained to follow the process we will be teaching; however, we would like to see certain products in certain formats ... all of which can be produced multiple ways.

To do this exercise, you will need to identify two things: (1) an organization with which to work (perhaps, your own); and (2) a process to define with that organization. If you have any problems with this, see Jim or Judy; we will try to find an organization and a process for your team.

If all contacts fail, Jim said that he has a few processes within PacSoft that could be defined and improved.

- Pick a simple, bounded process within an organization - it has to involve multiple roles (disciplines, groups).

  The two examples given below indicate the "level" to which you need to go (not very deep) and the "breadth" across the organization you need to cover (multiple organizations). (Note that these are written like "scenarios" not like a real process definition; that will become clear through the lectures and readings.)

  For example, managing a requirements (specification) change - customer calls, marketing fields the call and turns it into a requirement which is passed to engineering; they analyze it and feed impact back to their management and to marketing; ... design / code ... the independent test organization tests the code; defects are fed back into software engineering ... the ready-to-ship code is configured and handed off to the release group to cut the CDs; the CDs are passed to shipping which will pull the correct documentation, package all the contents, and ship the product to the customer.

  As another example, a customer calls the service organization with an urgent problem; the service organization verifies it is a critical software defect, files a defect report, and passes the issue to software engineering, requesting a fastpatch; software engineering evaluates the defect, the impact, and the resources needed to make the fix; software engineering management allocates resources to fix the defect; the defect is fixed and a fastpatch is created; peer reviews are held to verify it; service tests the fastpatch to verify it; the

© 1996 Judy Bamberger / James Hook

fastpatch is baselined (captured in a configuration management system); the defect report is updated to indicate a fastpatch was created and that it must be fixed in the next regular software release; a tape is cut, logged in the fastpatch tracking system, and shipped to the customer.

Your team is to turn in the following information at class #10:

(1) A process definition in all of the following representations:

- Value-added map ("context diagram")
- Time-x-role map
- EITVOX (only three process steps required to be done EITVOX; you may do as many as you want)

(2) A discussion of the team effectiveness, as described, to include:

- Team strengths, weaknesses
- Individual styles and how they contributed to the solution
- How decisions were made, how conflict was handled
- If you changed team members, discuss the impact
- Improvements you can make as a team, as individuals, for future team exercises
- Observations on overall team effectiveness
- How effectively you used your "together" time (e.g., meeting management)
- An indication of where you believe your team is on the Team Growth Model

You may also turn in any other "team stuff" you think would help us understand how you operated.

Your team is to present the following information at class #11:

- How your team did the homework; the team process and organization you used - the who, what, when, where, how, why of your team process (not effectiveness; that comes later!)

  [quick summary; be brief; ensure team consensus]

- What your team produced; a summary of the process your team defined - examples of all three representations (a summary of (1) above)

  [not expected to be a full presentation, simply a visual presentation to the class of what you produced for your partner organization and what you turned into the instructors]

- Observations on the reaction of your partner organization - to the process of defining processes, to the resultant process definition, to you as a team, etc

  [reflection, observations; ensure team consensus]

- Discussion of team effectiveness (a summary of (2) above)

  [short presentation to the class of what you produced as part of the assignment turned in the instructors]

© 1996 Judy Bamberger / James Hook

- A brief summary (team or individual members of the team) as to the utility of process definition techniques where you work (or have in the past, or would like to in the future).

  [briefly reflect on the techniques, the processes, the experiences, and your/team/organization's reactions]

- Be prepared to answer questions of clarification, curiousity, and envy from other students and the instructors on the above.

© 1996 Judy Bamberger / James Hook

End-Term Project: Process Improvement

To the Students:

This exercise is to be done as a team. Once again, "team" is defined as about three people. Effective meeting management and team collaboration skills are key to success of this exercise.

This exercise builds on mid-term process definition homework, class #12 (process improvement models) and other lectures, readings, and information about process improvement (in fact, the running theme throughout the course). Many of the remaining classes have no homework other than, "factor these concepts into the end-term process improvement homework"; we intend to provide information to help you address some of the questions we put to you. The homework is due, in writing, at class #16, Monday, 25 November 1996. Each team will present its results at class #17, Wednesday, 27 November 1996.

The assignment is to build on the process definition from "Week Five" homework, to work with your partner organization to:

- Identify metrics that can be used to measure the performance of the process today (to establish a baseline for improvement)
- Establish measurable goals for improving the process, and
- Develop an action plan to improve the process.

The information for the team to turn in is: the process improvement/action plan.

As individuals, you will look at individual and team dynamics and effectiveness. A list of topics to consider is included below.

The information for each individual to turn in is: a 2 - 4 page discussion of individual and team contributions.

You are not constrained to follow the models we will be teaching; however, we would like to see certain products with certain content, which is described below. We will provide additional templates for finished products (optional to use) at appropriate points in the remaining classes.

To do this exercise, you will need to reconfirm the willingness of your partner organization to continue collaboration. If you have any problems with this, see Jim or Judy; we will try to find a partner organization for your team.

You are encouraged to use the process improvement planning process and template introduced in class #12, unless your partner organization has a method and/or template of its own or you have a defined method and template of your own. If you do not use the method/template we teach, please let us know what you will be using, so we can ensure the necessary components will be covered.

© 1996 Judy Bamberger / James Hook

Be sure to provide us with a complete action plan - all sections. The following list provides some additional hints and references to help with some of the sections.

1.1. Problem Statement

A set of metrics that can be used to measure the performance of the process today.

- Describe any metrics used today to measure the performance of the process, and how they are collected, reported, and used.
- If none, describe what metrics could be used to establish a baseline so that the organization will know on what to base its improvement. Describe how they could be collected, reported, and used.

1.2. Vision after Success and

1.3. Goal Statement

A goal (or goals) that your partner organization would like to see for process improvement. You are encouraged to validate the goals with your partner organization, if they did not participate in their creation.

- Using the Goal/Question/Metric paradigm (class #13), develop the goal, questions, and metrics. Ensure that the metrics are quantifiable (either objectively, or subjectively; see homework/reading Weinberg, Volume 2, Chapters 8 and 9). For each of five metrics, describe how it could be collected, what is its scale (or value), any known or suspected data integrity issues, how it could be used to answer the questions to test achievement of the goal(s).
- Ensure that the goal(s) is a "SMART" goal (lecture #12), or discuss why they do not need to be, or the risk incurred if they are not SMART.

11. Roll-Out/Training Plan

Consideration for "self-sustaining improvement" (aka, institutionalization factors, discussed in classes #8 and #9)

ENABLERS for sustained improvement

- Commitment to Perform (policy, leadership needed)
- Ability to Perform (tools, training, resources needed)

ENFORCERS for sustained improvement

- Measurement and Analysis (how your proposed metrics will help demonstrate achievement of the goal, how the organization will know to what degree the organization is complying with the change)
- Verifying Implementation (progress/status reports to sponsoring management, independent review, etc)

Transition considerations for the new-improved process (e.g., information dissemination, newsletters, all-hands meetings, formal training, mentoring, brow

© 1996 Judy Bamberger / James Hook

bag lunches, etc). This needs to reflect sensitivity to the culture of your partner organization - how they learn and retain best.

© 1996 Judy Bamberger / James Hook

How to Get There

You are encouraged to involve your partner organization in this as much as they can be (or want to be). Since they are the ones who really practice the process being improved, they are likely to have many good ideas about how to improve it. Involvement can be: brainstorming during information/data collection; participation in the force-field analysis; reviewing intermediary products; etc.

You are encouraged to use one or more of the process improvement or problem solving methods discussed in class or offered as reading, unless there is another method with which you have much experience. (In this case, please consult with the instructors ahead of time to ensure the goals of this assignment are met.) Options include:

- The general problem solving method (and tools) from class #4 (remember to focus it on process improvement)
- The action planning techniques discussed in class #12
- The ODI method and tools from the readings for class #12 and class discussion
- The methods and tools described in *The Team Handbook*, chapter 5 (one of the readings for class #12)

Individual and Team Contributions

As individuals, please provide a discussion of ***your view*** of your contribution to your team's effectiveness throughout the course, to include:

- Your style and how it contributed to the strength of the team
- Observations on areas where you would like to improve in team activities (styles you might want to try; roles you might want to play, etc)
- Techniques and skills you used to keep the team moving forward, to resolve conflicts, to be creative, etc
- How effectively you used your "together" time (e.g., meeting management)
- An indication of where you believe your team is on the Team Growth Model
- Effectiveness of team activities in reinforcing the key themes of this course
- Lessons you have learned from your class-team that you can take into your work-place-teams

© 1996 Judy Bamberger / James Hook

Team Presentation

Your team is to present (30-45 minutes; depends on the total number of teams) the following information at class #17:

- How your team did the homework; the team process and organization you used - the who, what, when, where, how, why of your team process

  [quick summary; be brief; ensure team consensus]

- What your team produced; a summary of the action plan

  [not expected to be a full presentation, simply a visual presentation to the class of what you produced for your partner organization and what you turned into the instructors]

- Observations on the reaction of your partner organization - to the process of improving processes, to the resultant process improvement plan, to you as a team, etc

  [reflection, observations; ensure team consensus]

- Be prepared to answer questions of clarification, curiousity, and envy from other students and the instructors on the above.

© 1996 Judy Bamberger / James Hook

FROM MARCH'S
BYTE; FYI.

MEMORANDUM OF CALL

TO:

☐ YOU WERE CALLED BY— ☐ YOU WERE VISITED BY—

OF *(Organization)*

☐ PLEASE CALL → PHONE NO. CODE/EXT. _____ ☐ FTS

☐ WILL CALL AGAIN ☐ IS WAITING TO SEE YOU

☐ RETURNED YOUR CALL ☐ WISHES AN APPOINTMENT

MESSAGE

RECEIVED BY	DATE	TIME

63–109

☆GPO : 1981 O - 341-529 (117)

STANDARD FORM 63 (Rev. 8–76)
Prescribed by GSA
FPMR (41 CFR) 101–11.6

record and file locking, semaphores, shared memory management, and hardware error recovery. The version of XENIX that you can buy depends on your computer. XENIX 3.0 runs on the IBM PC AT, while XENIX 2.3 runs on the Altos, Tandy, and other 16-bit multiuser computers.

The Real RSA Algorithm

Charles Kluepfel's article ("Implementing Cryptographic Algorithms on Microcomputers," October 1984, page 126) is not based on the real RSA algorithm but on Donald Knuth's version of it. Knuth uses the exponent 3 to encode a message, but the full RSA allows any exponent that does not share any prime factors with $(p-1)*(q-1)$. Instead of having to ensure that messages are greater than the $n\hat{}(1/3)$, one can choose any encoding key s such that $2\hat{}s>n$ and then be sure that all messages except 0, 1, and $n-1$ are thoroughly encrypted.

The full RSA system also allows the decoding key to be chosen for special properties and the encoding key to be deduced from it; for instance, the decoding key may be kept short (15 digits or so) or close to a power of 2 for easier computation when the recipient of messages has less computing power than the sender.

I'm not sure why Knuth's version is different; perhaps his knowledge of RSA was based on an early version, before the main paper was published: CACM, volumes 21 and 22, pages 120–126 (1978).

I have tried running Kluepfel's example on our own Big Integer BASIC interpreter on a 3-MHz Z80-based CP/M machine, with the following program:

```
100 INPUT N,D
110 INPUT MS
120 CD=MS^[N]3 : PRINT CD
130 MT=CD^[N]D
140 IF MS=MT THEN PRINT "OK" :
    PRINT : GOTO 110
150 PRINT "**ERROR**";MT
```

Apart from problems with a misprint in listing 9 (a spufious "1" in "182818218" in the first two occurrences of MS), the program ran first time. It took a second or so to encode and 115 seconds to decode. Our factorization program in BASIC took 2.5 minutes to factor BYTE's telephone number: 13 * 4703 * 98779 = 6039249281. No doubt the IBM PC version will be faster.

Martin Kochanski
Speldhurst, Kent, England

Charles Kluepfel replies:

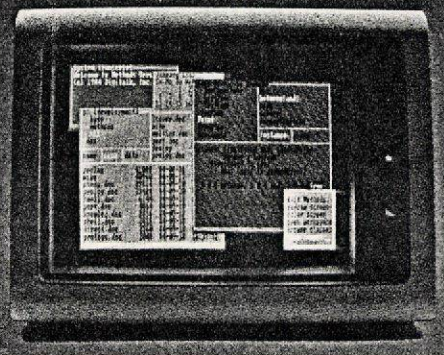
At the time I wrote the program and article, all descriptive references that I saw to the RSA system used the power 3, including Knuth, who in fact referenced the same article Mr. Kochanski mentioned. As Knuth provided an unambiguous description, I felt that it was the same as in the RSA reference, and I did not seek that source. However, the Knuth description is, indeed, based on the real RSA algorithm, as a particular instance of using 3 as the encoding power.

Referring now to that main paper, wherein the power in question is denoted by e (as opposed to s in Kochan-

(continued)

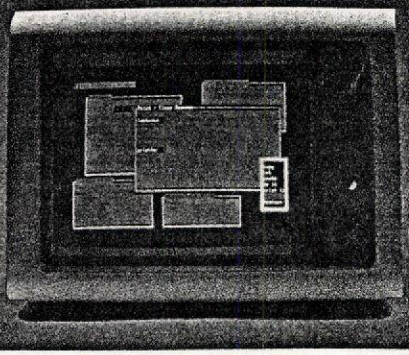
A Classy Instance of Smalltalk

If your object is easy programming, our message is . . .



Methods

A complete object-oriented program development environment with Smalltalk-80™ language compatibility.



It's powerful, fast and fun!

For IBM PCs and compatibles with 512K bytes RAM using PC-DOS or MS-DOS.

DIGITALK, INC.

5200 West Century Boulevard
Los Angeles, California 90045 (213) 645-1082

Smalltalk-80 is a trademark of Xerox Corporation. MS-DOS is a trademark of Microsoft, Inc. PC-DOS is a trademark of IBM Corporation.

YES! Please send me a copy of **METHODS** for $250.

Check____ Money Order____ Visa____ Mastercard____

Card # _____ Exp. Date____

Name _____

Address _____

City/State/Zip _____

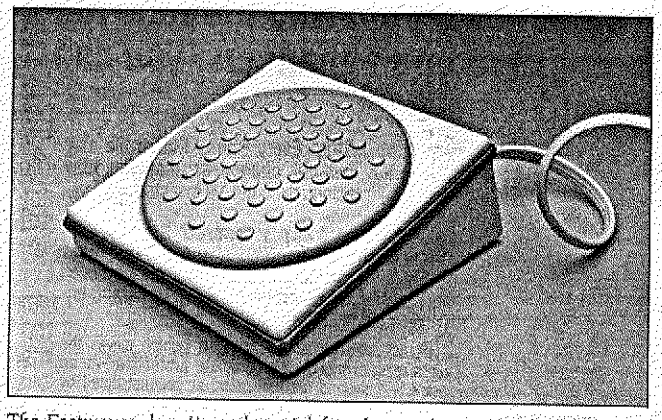
Telephone ( ) _____

Signature _____

California residents add 6% sales tax. Outside U.S.A. add $15.00

Inquiry 116

Footmouse Frees Your Hands



The Footmouse doesn't need special boards or software.

Versatron is shipping the Footmouse, a foot-operated mouse for microcomputers. The manufacturer notes that the primary advantage of the Footmouse is that it frees both your hands for data input.

Footmouse reportedly works with any software package that uses a cursor. It emulates the keyboard cursor functions, yet it does not interrupt normal cursor operations. Footmouse plugs between the keyboard and the computer, requiring neither special boards nor software support.

Presently available for the IBM PC and IBM PC-compatibles, versions of the Footmouse for the Apple IIe, Macintosh, IBM PC XT and PC AT, Ivy, Compaq, and RS-232C terminals will be available shortly. The suggested list price is $225. Contact Versatron Corp., 103 Plaza St., Healdsburg, CA 95448, (800) 443-1550; in California, (800) 435-1550 or (707) 433-8244.

Inquiry **609**.

Multitasking, Multiuser DOS Runs with MS-DOS

A multitasking, multiuser operating system for 8086/8088 microcomputers running MS-DOS has been introduced by FORTH Inc. Called polyFORTH II, this operating system gives you the ability to run multiple terminals, unlimited tasks, and concurrent printer operations. The environment that polyFORTH II creates is said to be suitable for such interactive, real-time computer-control applications as robotics, data acquisition, image processing, and process control.

Any number of asynchronous processes running concurrently are supported by polyFORTH II. A company spokesperson reports that polyFORTH II does not impose a cap on the amount of users supported, although this is subject to hardware constraints. Further, the spokesperson notes that polyFORTH operates at reasonable speeds, the rate of which is dependent on the number of processes running.

Tasks can be assigned private partitions, or they may execute shared, reentrant routines. Active tasks require as little as 100 bytes of memory, and context switches need only 14 machine-language instructions.

Several configurations of polyFORTH II, reflecting increased capabilities and support services, are available for MS-DOS computers. Level 3, which costs $600, includes the operating system, a FORTH turnkey compiler, assembler, editor, mathematics library, database support system, utilities, and source code for all but the nucleus.

Priced at $3200, polyFORTH II level 4 comes with all the capabilities of level 3 as well as full system source and the Target Compiler, which is capable of generating applications that can be embedded in ROM or recompiling polyFORTH itself.

All polyFORTH II disks are compatible with MS-DOS, and its FORTH blocks are maintained in data files. Contact FORTH Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254, (213) 372-8493.

Inquiry **610**.

Methods for Smalltalk Programming

Methods is a Smalltalk program-development system for the IBM PC and compatibles running under DOS versions 2.0, 2.1, or 3.0. Fully compatible with Xerox's Smalltalk-80 language, Methods includes nearly 100 classes, which are programming tools that define the structure and behavior of abstract data types such as integers and points.

Smalltalk, an extensible, object-oriented language, is suitable for simulation and graphical user interfaces. For a broader discussion of Smalltalk, see the August 1981 BYTE.

Methods gives you access to most of the source code from which it is built. It has more than 2000 routines, or methods, that you can browse through, put to use, or modify. Primitive methods can be implemented in assembly language.

The user interface features a character-mapped display, pop-up menus, and extensive use of color (monochrome displays are supported). Your cursor keypad is used as if it were a mouse. Methods also comes with a system transcript, file editor, and a window for debugging.

Methods requires 512K bytes of RAM and a pair of 360K-byte disk drives or a hard disk. Two manuals are supplied. The suggested price is $250. Contact Digitalk Inc., 5200 West Century Blvd., Los Angeles, CA 90045, (213) 645-1082.

Inquiry **611**.

Peripheral Boosts the Mac's Versatility

MacEnhancer from Microsoft lets you add three different peripherals to Apple's Macintosh. Requiring a single Macintosh RS-422 port, MacEnhancer gives you two RS-232C serial ports and a parallel printer interface.

MacEnhancer arrives with drivers for a number of popular dot-matrix and daisy-wheel printers and with terminal-emulation software for accessing information services and bulletin boards. Its list price is $249. For further information, contact Microsoft Corp., 10700 Northup Way, POB 97200, Bellevue, WA 98009, (206) 828-7400.

Inquiry **612**.

(*continued on page* 435)

Smalltalk Grows Up

Thanks to a boost from IBM and Microsoft, and a growing set of support tools, Smalltalk is finally beginning to sound good to IS.

BY JEFF MOAD

If object-oriented programming (OOP) technologies are the wave of the future, why hasn't Smalltalk, the granddaddy of OOP languages, been more successful?

After all, Smalltalk, which was developed at the Xerox PARC research labs in Palo Alto in the mid-1970s, was one of the first languages to reject the operator/operand, linear style of more conventional programming languages. Instead, Smalltalk uses self-contained data structures called objects, which programmers can combine and reuse in their applications. Its proponents said Smalltalk would significantly improve programmer productivity and make long-term program maintenance and enhancement easier.

But Smalltalk never really took off. IS applications development managers have tended to view it as a fringe language, weak on performance and lacking critical support features, such as stable database interfaces and well-established development methodologies. Thus, IS managers have been reluctant to retrain their COBOL programmers to use Smalltalk. ParcPlace Systems, a spin-off from Xerox PARC, and a few small consulting firms were pretty much alone in trying to convince IS to take Smalltalk seriously.

A Small Surge

All that may be changing, however. A number of IS organizations are moving beyond experimenting or prototyping applications with Smalltalk and are beginning to develop critical applications. Several small vendors have recently entered the Smalltalk market with products that make the language easier to use and more productive for large applications. And a ParcPlace competitor has even emerged. Digitalk Inc. of Los Angeles is now selling lower cost versions of Smalltalk targeted at OS/2 and Windows users.

Smalltalk is not about to replace COBOL, but it is finally maturing into a viable choice in application development, especially for users looking for a tool to speed development of advanced graphical user interfaces (GUIs) in client/server applications.

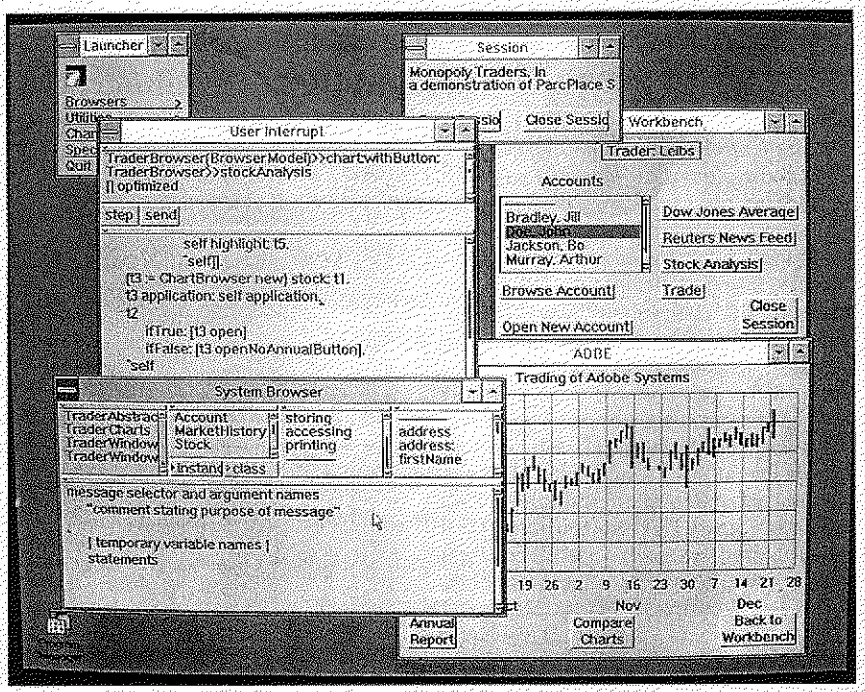
That's Smalltalk's function at Texaco Inc.'s oil exploration and land management unit in Houston. According to Texaco IS manager Dennis Samoska, the company wanted to rewrite two large mainframe applications, replacing dumb terminals with Windows-based PCs and workstations. The PCs and workstations would give users easier access to host data via advanced GUIs. After evaluating several computer-aided software engineering (CASE) tools, Samoska's unit decided to prototype and implement its new applications using ParcPlace's Objectworks\Smalltalk. Now under development, the applications will access Texaco's existing DB2 host database via an application programming interface (API).

Users like Samoska say Smalltalk takes much of the headache out of creating applications that use such GUI standards as Windows or OS/2 Presentation Manager. Rather than require programmers to learn and write to complicated GUI APIs, Smalltalk uses reusable class libraries that can link applications to the APIs. Developers can then work with an easier-to-use set of interface-building tools.

As a dynamically compiled language built on reusable objects and a virtual interface that uses machine-independent, intermediate code, Smalltalk is also easily portable between the platforms it supports. Texaco, for example, is developing its new Smalltalk applications on Sun workstations and is running them on PCs.

Help From The Big Boys

Smalltalk is also benefiting from recent recognition from IBM and Microsoft Corp. Recognizing that Smalltalk could help users write more OS/2 PM applica-



SMALLTALK\OBJECT from ParcPlace Systems can be used to build portable applications that run under several graphical interfaces.

tions, IBM endorsed the environment late last year. It signed licensing agreements with both ParcPlace and Digitalk. IBM is currently attempting to integrate Smalltalk and object-oriented programming capabilities into its AD/Cycle CASE architecture. If developers at Big Blue's Programming Systems Lab in Cary, N.C., are successful in integrating Smalltalk into AD/Cycle's information model, there's a good chance IBM will take the next step and declare Smalltalk a Systems Application Architecture (SAA)-approved language, says Cliff Reeves, manager of Common User Access for IBM.

☐ THERE ARE STILL SEVERAL MISSING CHAPTERS IN THE SMALLTALK STORY.

Not to be left behind, Microsoft is moving Smalltalk into the Windows environment. The company is working closely with ParcPlace to build low-level program-to-program messaging links into future versions of Windows and OS/2 that could help programs written in Smalltalk and other object-oriented environments such as C++ more easily share objects.

Eiffel Tower Of Babble

Public support of Smalltalk by IBM and Microsoft has given a boost to some IS managers trying to sell their bosses on the advantages of object-oriented programming and Smalltalk. "It really helps," says Phil Hartley, principal technologist in the Advanced Technology Group at American Airlines Inc.'s SABRE Computer Services. Hartley's group is currently evaluating Smalltalk along with other object-oriented and GUI-building tools such as C++, Eiffel and Object COBOL. Smalltalk currently has the strongest set of support tools, says Hartley.

Indeed, thanks to a growing number of new Smalltalk independent software vendors (ISVs), users can now choose from several sets of tools that extend

Helping Smalltalk Get Big

For more information on the companies mentioned in this article and their products, circle the appropriate numbers on the Reader Service Card.

Acumen Software
Berkeley, Calif.
Circle No. 400

Digitalk Inc.
Los Angeles
Circle No. 401

Easel Corp.
Burlington, Mass.
Circle No. 402

Gupta Technologies Inc.
Menlo Park, Calif.
Circle No. 403

Instantiations Inc.
Portland, Ore.
Circle No. 404

Micro Focus Inc.
Palo Alto
Circle No. 405

Microsoft Corp.
Redmond, Wash.
Circle No. 406

ParcPlace Systems
Mountain View, Calif.
Circle No. 407

Sybase Inc.
Emeryville, Calif.
Circle No. 408

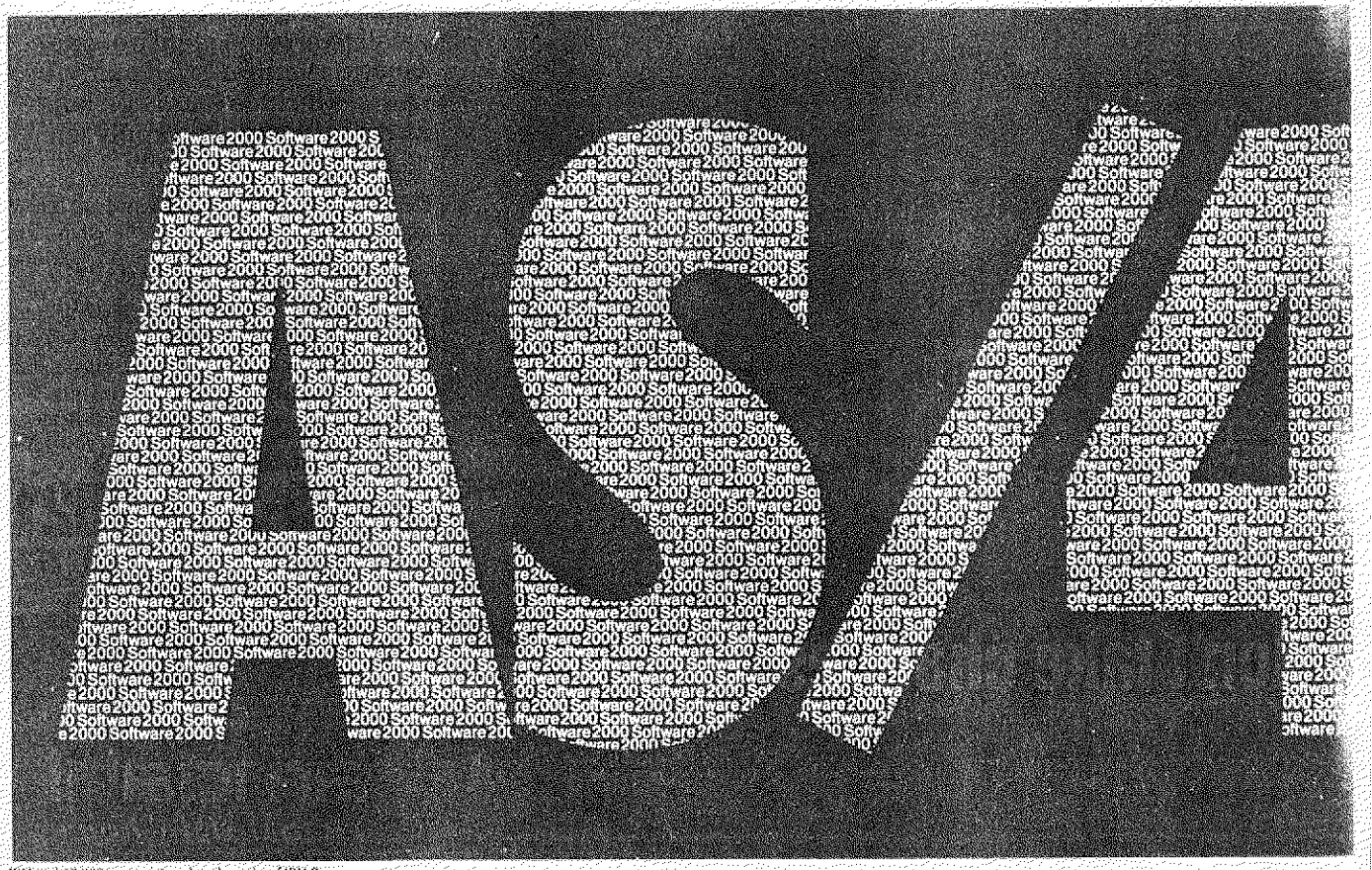
Synergistic Solutions Inc.
Highlands, N.J.
Circle No. 409

Tigre Object Systems Inc.
Santa Cruz, Calif.
Circle No. 410

Whitewater Group
Evanston, Ill.
Circle No. 411

Source: DATAMATION

The Closer You Look At Software For The AS/400,



IBM and AS/400 are registered trademarks of IBM Corp.

Smalltalk's functionality and ease of use. Last year, Highlands, N.J.-based Synergistic Solutions Inc. started shipping its Smalltalk Platform for Integrated Computing Environments (SPICE), which includes a set of class libraries. SPICE helps link applications written in Objectworks\Smalltalk or Digitalk's Smalltalk V with the Sybase/Microsoft SQL Server and with NetBIOS or DEC-Net networking protocol interfaces. Through the Sybase database gateway, Smalltalk applications can also access DB2, Rdb and other database management systems.

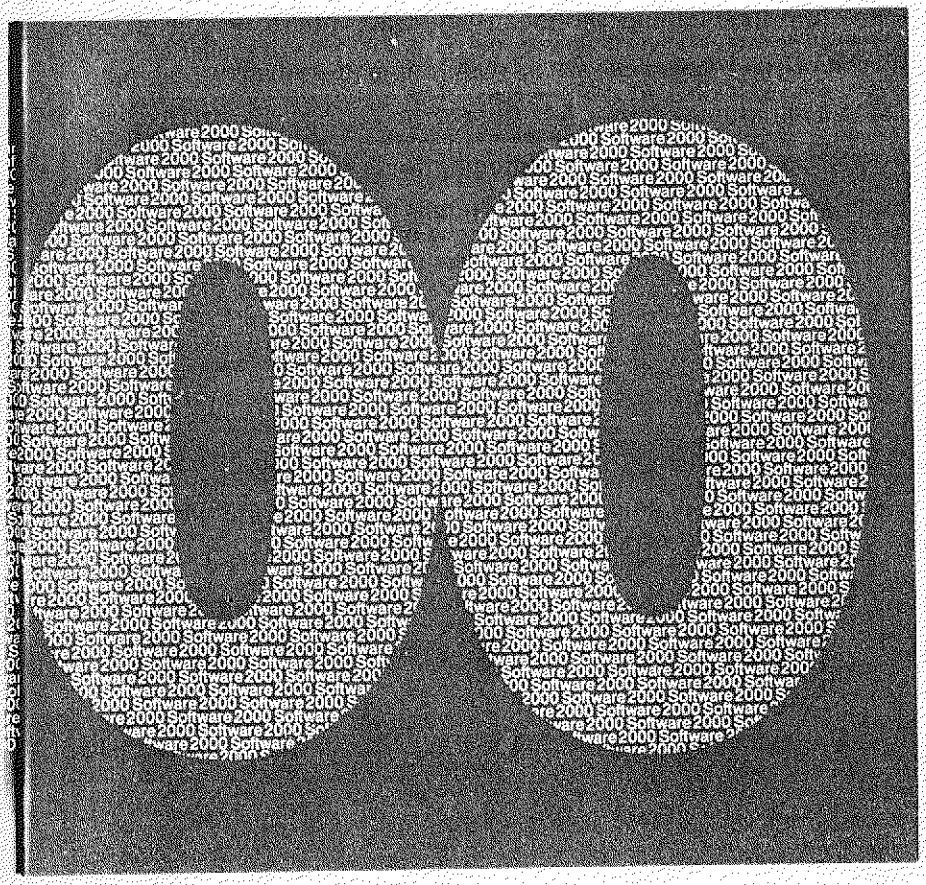
Similarly, Instantiations Inc. of Portland, Ore., recently started shipping what it calls the Application Organizer Plus, a set of tools that helps large Smalltalk development teams with version management and code reuse. And at least two other small vendors, Acumen Software of Berkeley, Calif., and Tigre Object Systems Inc. of Santa Cruz, Calif., are shipping tools aimed at helping Smalltalk developers create GUI-based applications on PC, Macintosh and UNIX platforms even more easily.

Even one mainstream language vendor—Micro Focus Inc.—is getting into the Smalltalk environment. The vendor of COBOL development tools recently agreed to bundle Digitalk's Smalltalk V into its Cobol Workbench to be used to build OS/2 PM interfaces.

There are still several missing chapters in Smalltalk's story, however. Observers, note that there is still no widely accepted development methodology for Smalltalk or for any other object-oriented environment. In addition, many users are still making the transition to the relational model and structured programming techniques. "Most [IS developers] still don't know what to do with objects. They're still traumatized from making the migration to the RDBMS," says Natasha Krol, application program director at the Meta Group in Stamford, Conn. Smalltalk also faces increasingly stiff competition not only from other object-oriented languages such as C++ but also from new GUI-building tools, such as Easel from Easel Corp. and Actor from Whitewater Group.

Still, many observers see Smalltalk gaining maturity. Says Stuart Woodring, an analyst with Cambridge, Mass.-based Forrester Research Inc.: "Continued strong support from IBM and others could help push it over the top."

The Clearer The Answer Becomes.



Of the thousands of business solutions for the AS/400, Software 2000 is clearly the best. We've been dedicated to the AS/400 platform since day one. And our cooperative research and development relationship with IBM ensures that our business solutions are available with the very latest AS/400 enhancements.

Our Software 2000 Series includes a complete range of integrated financial, human resources, environmental and distribution software that provides mainframe functionality with PC ease-of-use. We also offer a suite of PC-based cooperative processing products that are designed to help you better plan for the future by bringing your organization's most critical information to the desktop. For an even closer look at what are clearly the best business solutions for the AS/400, call Software 2000 at (800) 388-2000.

Software 2000

The AS/400 Business Solution.

Circle 2 on Reader Card

IEEE Computer Graphics and Applications

December 1984 Volume 4 Number 12 (ISSN0272-1716)

ARTICLES

Cover: Damon Rarey of Aurora Systems tells the intriguing story of this month's cover in "About the Cover" on page 4.

Cover design: Jay Simpson

DEPARTMENTS

Published by the IEEE Computer Society

IEEE CS Membership Application, p. 12
Change-of-Address Form, p. 79
Reader Service Cards, p. 97

Even with limited resources, it is possible to improve man-machine communication by employing the same graphic design principles used in large commercial systems.

Corporate Identity for Iconic Interface Design: The Graphic Design Perspective

Aaron Marcus
Aaron Marcus and Associates

As computer systems become more sophisticated, they must remain friendly, comprehensible, and effective to continue to appeal to users. A crucial factor in all three of these desirable attributes is the quality of communication between user and machine.

The quality of communication is included in all of the features commonly felt to be found in an effective system: low cost, sophisticated functionality, friendly interfaces, and good service.[1] This common thread of communication affects the long-term cost of the system by reducing the nonproductive time during training and use of a computer system, by providing the means for the user to take advantage of the system's functional power, and by enhancing service quality when users understand easily how to achieve their goals. Communication can be understood to take place through three "faces": *outerfaces* (presentational and analytic displays), *interfaces* (user-machine command/control and documentation dialogue), and *innerfaces* (programming and maintenance environments).[2]

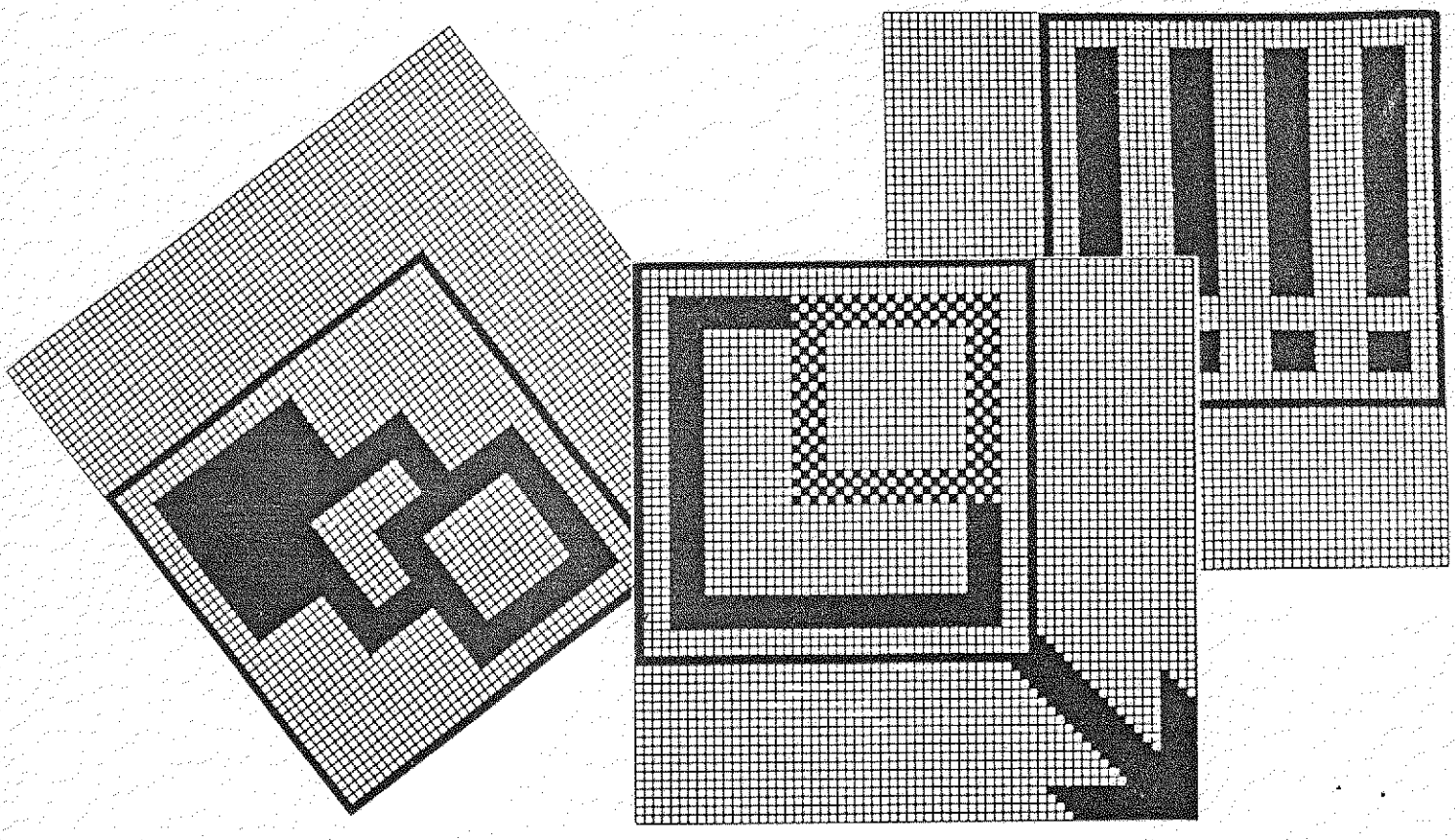
Good communication generally can be achieved when the following conditions are present:

1. Simplicity—major parts of the system are few in number or are hierarchically organized.
2. Clarity—the parts of the system are evident.
3. Familiarity—the parts of the system remind the user of things already known.
4. Integrity—the system is an ordered sum of its parts.
5. Consistency—what the user knows of one part helps in other parts of the system.
6. Reliability—the system responds to the user in a trustworthy manner.
7. Responsiveness—the interactive replies of the system are quick, polite, and helpful.[3,4]

There are no simple rules for achieving good communication because all of the contributing factors interact with each other. Since the subject is a broad one, this article focuses on screen design issues from the graphic design perspective.

Human-computer screen interfaces are often an unorganized series of frames that are never seen as an entity. Once the interfaces are viewed as a whole, it becomes apparent that one can speak of the "corporate identity" of an interface—that is, the consistent, articulatable step-by-step application of typography, symbolism, color, spatial layout, and approach to sequencing/animation that characterize a particular system. The term "corporate identity program" is well known in the graphic design field, which has applied the approach to the traditional areas of stationery, vehicle identification, signage, and other forms of complex communication. This approach can now be extended to the design of screens, especially for high-resolution, iconic, multiwindow interfaces.

This article is based on an earlier article, "Iconic Interface Design," that appeared in the *Proceedings of Nicograph 83*, Tokyo, Japan, 1983, pp. 103-123. The article originally appeared in the *Proceedings of the Fifth Annual Conference of the National Computer Graphics Association, Vol. 2: Tutorials*, (Anaheim, California, May 13-17, 1984), pp. 468-479, and is reprinted in revised form with the permission of the National Computer Graphics Association.



Graphic design and corporate identity programs

The systematic appearance of typography, symbolism, color, layout, and sequencing constitutes a visible language. Corporate identity programs establish the system and record it within graphic design standards manuals, which, in effect become the source code for a graphic design algorithm. (Design manuals are well understood in the graphic design field; in fact a design manual for design manuals has even been written.[5]) Without reproducing a manual here, we can delineate the main points of a typical manual's conventions. Those conventions relevant to high-resolution, iconic, multiwindow screen design appear in the subsections below.

Typography. Variations in typefaces should be limited to one or two type families. Many of the leading corporate graphics manuals in the business community have traditionally emphasized a few well-documented type fonts. These have proven their legibility, their flexibility of display in small text sizes and large display sizes, and their availability in many styles. The more popular typefaces include: Baskerville, Caslon, Garamond, Helvetica, Times Roman, and Univers.[6]

Sometimes two type families are combined. A typical combination of two faces might be Helvetica and Times Roman or Univers and Times Roman, that is, a contrast between sans-serif and serif typefaces. Type size is also limited: three sizes (or a maximum of five) suffice for all distinctions such as footnotes, titles, headers, and figure captions. Wherever possible, simple, direct changes in size of 2:1 or 3:1 should be used to distinguish the levels of titling sizes.

Type appears in columns, usually one to three columns per screen, with 40 to 60 characters per column. Upper- and lowercase letters should be used wherever possible because they are more legible. Their legibility is due to the greater differentiation of letter shapes which contribute to the overall shape of the word. Reading is accomplished by recognition of word shape as much as by the identification of individual letters. All capital letter settings of text may be used for brief titles or for emphasis, but their extensive use can slow reading speed by as much as 13 percent.[5] The space between words should be approximately one quarter the width of a capital M for variable width fonts, while the space between lines may vary according to the design of the type font. Generally, sans-serif letterforms like those of Helvetica require a small amount of extra space between the baselines of the text. For maximum legibility the line spacing should produce spaces between lines greater than the amount of space between words.

The lines themselves may be unjustified (ragged right); there is no noticeable difference in their legibility.[7] The differences in line length of unjustified text can contribute to the visual interest of the screen, but care must be taken to avoid strong, recognizable shapes in the pattern produced by the ends of lines. The columns of text themselves should be separated by a width at least equal to two word spaces for variable width characters. For many screens a

layout of three columns per screen for text settings or one wide column on the right with a narrow column on the left for marginalia is useful.

Typical choices for type slants include roman and italic; typical type weights are bold, medium, and light, although it is questionable whether light typefaces of small text sizes will be legible on most current high-resolution display screens. Typical width variations for type in text normally include condensed, regular, and extended; but for most screen presentations, regular widths suffice. A useful set of style variations would then include medium roman, medium italic, bold roman, and bold italic for content variations in normal text settings.

Symbolism. According to the language of semiotics, the science of signs, signs may be iconic (representational) and symbolic (abstract).[8,9] Symbolism as used here refers loosely to all nonverbal signs: illustrations, photographs, diagrams, pictograms, etc.

The concept of corporate graphics implies that all images are designed to meet their unique communication needs, while being adjusted to produce a visual consistency throughout the system. This combined approach can be achieved by the use of a constant scale, limited size variations, the orientation of figures with respect to text, limited use of colors, limited variation of line weights, and the treatment of the borders for figures or pictograms. These visual themes help to establish recognizability, clarity, and consistency just as verbal or linguistic techniques applied to text help to promote simplicity, clarity, familiarity, integrity, and consistency.

One area receiving considerable attention in the corporate graphic design community is the design of logos, ideograms, and pictograms to communicate the concept of a total business entity to guide consumers in the use of a company's products (packaging signage), or to guide visitors through a company's architectural environment (architectural or urban signage).[10] In some fields, such as transportation, standards have emerged.[11] Here the symbolism is often characterized by a functional elegance: unnecessary variations of curvature, line thickness, shape, color, and number of parts are avoided. Many of the most widely recognized company trademarks are models of good design, a fact which makes them suitable for appearance in many sizes and a variety of display media.

Color. The use of color in computer graphics has often emphasized too many colors, even when only a few were available.[12] The corporate graphics approach to color emphasizes the selection of a limited set of well-chosen colors that meet the criteria of production, the needs of the content, and the preferences or limitations of the viewers. These colors are used repetitively to maintain consistency across content areas and across different display media.

The colors chosen by a company can be a primary feature in achieving recognition by its public. If the set of selected colors is sufficiently large, a designer can use them in many forms of informational as well as marketing graphics. Some companies choose color schemes that are very simple, while others choose nonstandard, more subtle, muted (low chroma) colors as their unique color identity.

Layout. The approach to spatial organization characterizing corporate graphic design derives from the European constructivist artistic movement of the early twentieth century. As the approach found its way into the formative years of the international style of corporate graphic design during the 1950's, it stressed an articulate, systematic method of assigning areas for text and illustration as well as the background field or format. Whenever possible, visual references were made to a series of strong, easily recognized proportions that have been used since classical times:

1:1.000, the square
1:1.414, the square-root-of-two rectangle
1:1.618, the golden rectangle
1:1.732, the square-root-of-three rectangle
1:2.000, the double square

When multiple columns of text or images are used, a designer can create more interesting and lively compositions of text and illustration. A typical screen layout might propose three equal-width columns or one narrow and one wide column. Typically, a large space is left at the top of the screen for important titling or illustrations. The layout grid forms the basis for regulating the varied groupings of text and images. This grid is a series of horizontal and vertical lines that define certain areas of the screen for the positioning of titles, text, or illustrations. The grid also determines the extent or size of these three elements. In this way the approach builds a visual consistency into every possible layout.[7]

While never visible in its entirety, the grid is always present through its effects on visual elements of the composition. Even empty space is a meaningful part of a gridded layout because it can restate the grid's subdivisions of space. The concept of the spatial layout grid emphasizes diversity within limited constraints. The grid is a means of establishing recognizable order and hierarchy within a complex problem of location, shape, size, and content.

Sequencing. Traditionally animation and kinetic movement have not played major roles in corporate design programs because the access to control has been lacking in display media. Where temporal design is possible, the corporate graphics approach again stresses simple, clear, modular temporal constructs. This might apply, for example, to the regular appearance or disappearance of items or the overall dramatic narrative.

Case studies

Several office automation systems have appeared that display characteristics of the corporate identity approach to the design of the human-machine interface with varying degrees of completion. These microprocessor systems are supported by high-resolution bit-mapped screens. This article examines three black-and-white systems: the Xerox Star, the Apple Lisa, and the Intran Metaform.

The Xerox Star. The Xerox Star system (see Figure 1) appeared in 1981; it emerged from research efforts at Xerox PARC. Based on published documents, a number of the fundamental design principles for the Star interface

are known.[4] In creating a system that promoted familiarity and friendliness through the simplicity, coherency, and consistency of its interface, the designers sought to develop a conceptual model of the system in the mind of the user that was communicated through the visual features of the interface. The corporate approach to communication strives for exactly this method—to embody functions and features into an easily grasped and easily learned system. The Xerox team articulated the desktop metaphor on the screen to represent activities generally handled at a desk and carried it through in the visualization of all desklike functions and activities. This was accomplished through visual objects called "icons," which have properties that are summarized for the user in easily displayed and edited "property sheets."

By establishing global commands with consistent meanings throughout the Star interface, its designers were able to develop another systematic aspect of the display. Several other methods were used to establish consistency: Editing is accomplished through a single paradigm of operation, whether one is editing text, graphics, files, the desktop itself, property sheets, or even programs. Retrieving information always takes place through the paradigm of databases, and creating new objects is always achieved through the paradigm of copying.

By reducing the number of parts in the system and minimizing redundancy, Star designers were able to achieve large-scale simplicity. One important aspect of this simplicity is modeless interaction. For example, the keys of the main keyboard are used for typing only, and special keys are used solely for functions. Clearly separating these keys reduces the short-term memory requirements of the user.

The Star organizes the desktop screen into a space for 154 icons centered on fixed locations, each one inch square or 72×72 pixels. The squarish icons tend to fill up their allotted space and use small changes in their edges to communicate the different meanings. For example, a small corner tipped down on a page represents a file, while a small extension of the top edge of a folder represents a collection of files. Screen buttons differ in the drawing of their corners: sharp, square corners represent items of data or characteristics on property sheets; rounded corners represent screen buttons and titling within the top band of windows.

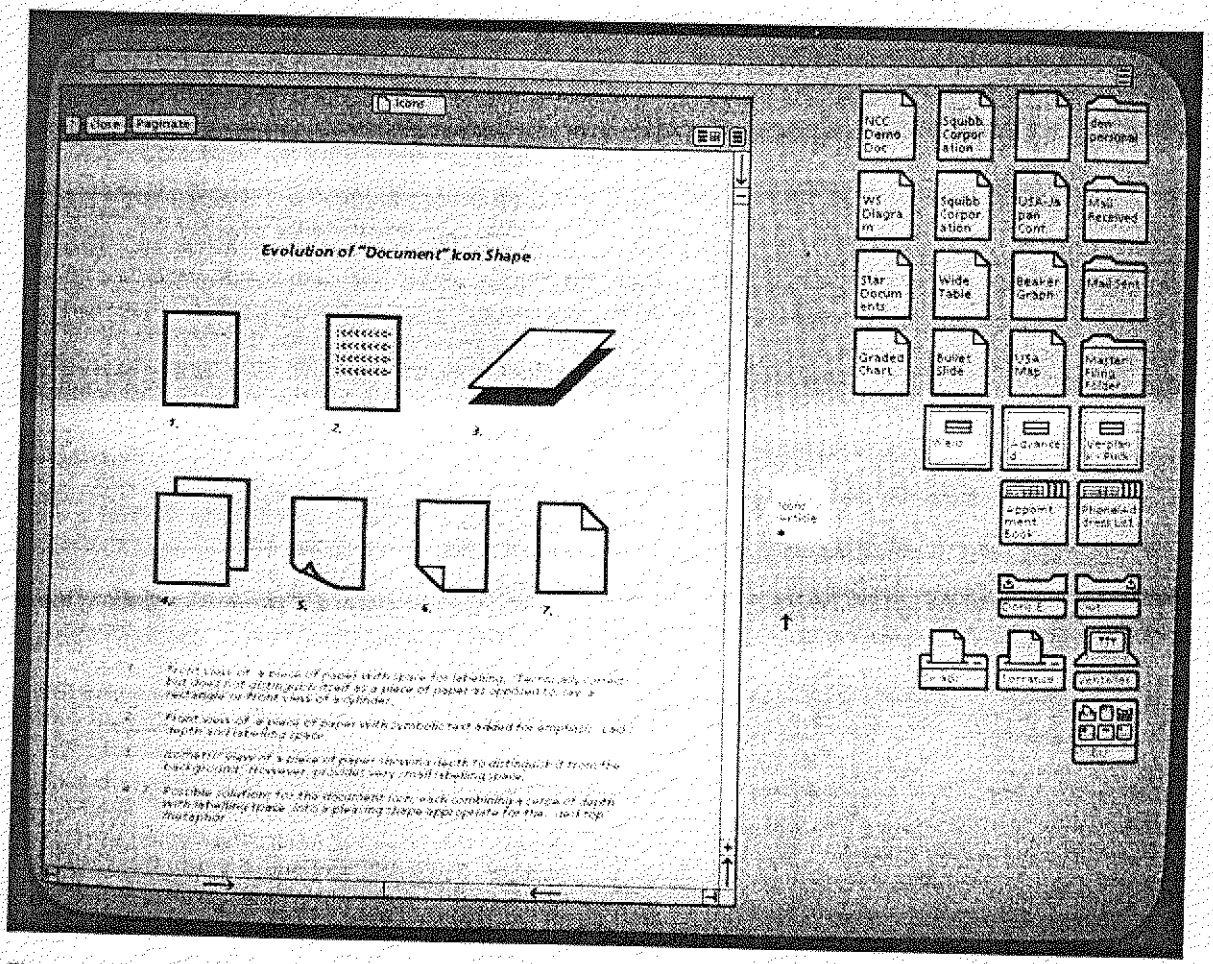


Figure 1. Typical screen layout for the Xerox 8010 Star Information System. Visible are some of the iconic screen buttons. The icons are recognizable images of familiar office objects, such as documents, folders, file drawers, and "in" and "out" baskets. The window is at left with its title border at the top and additional symbols appearing in the borders at the right side and the bottom. (Photograph courtesy of Xerox Corporation.)

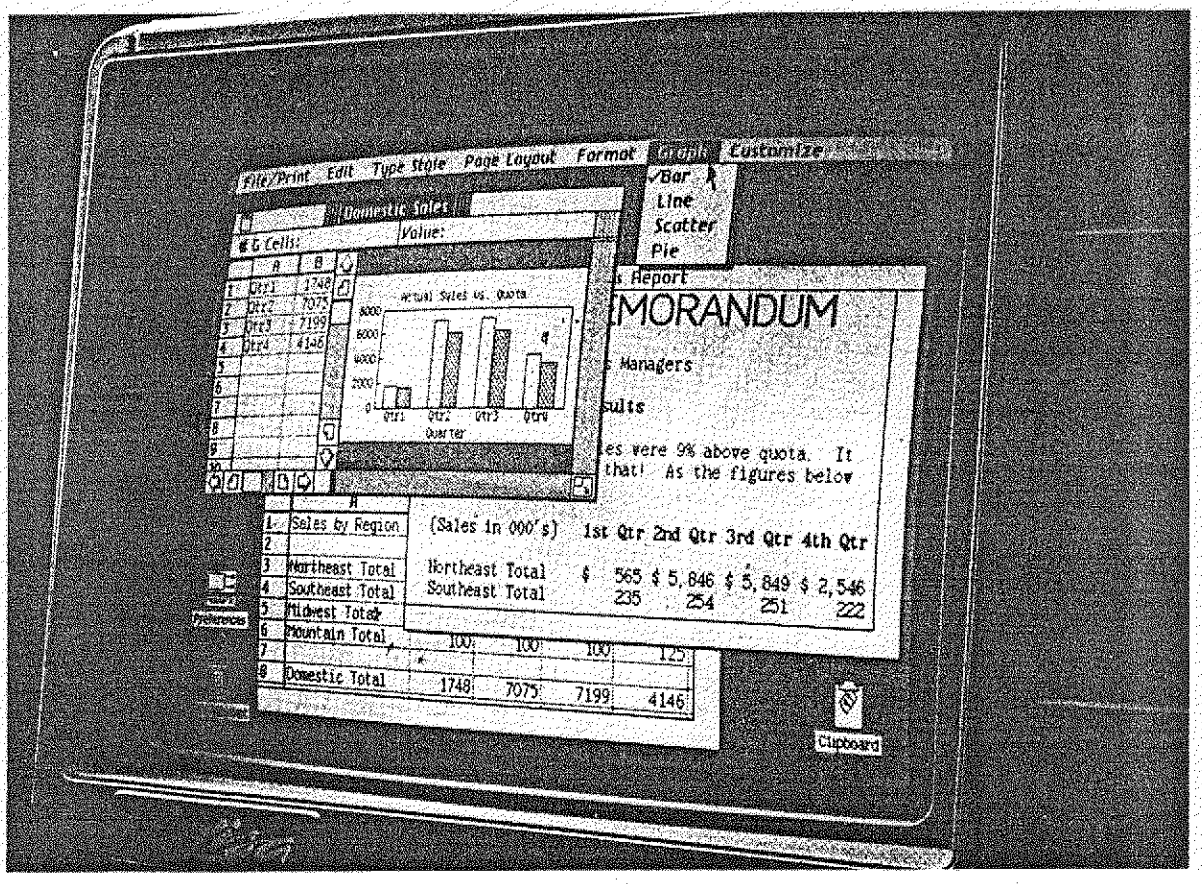


Figure 2. Typical screen layout for the Apple Lisa system. Visible are some of its iconic screen buttons, multiple overlayed windows, title bar, and descending submenus. Each window exhibits window border symbolism.

The Star is a black-and-white system. Color issues concern how white, black, and gray are used. The desktop screen itself is approximately 50 percent gray. The dark outlines of the icons and the dark outlines of windows show up clearly against it. The interior areas of the icons and windows are white to contrast with the background. The top border of subwindows appears as a higher resolution gray to distinguish itself from the two other screen elements: window contents and desktop. Black is used for reversing screen buttons or icons that have been selected; the selected item reverses within the old area, and a thin white rule surrounds it. All of these details represent a particular systemwide approach to the treatment of color.

The icons of the Star appear in a fixed grid layout measuring 14 units wide by 11 units high. Another fixed-grid aspect of the interface is the top border of a window: It must accommodate two rows of window titles and local screen buttons. The remaining grid features are the right side and bottom borders of windows: They must accommodate screen buttons for scrolling within the window.

A particularly characteristic feature of the systematic approach in the Star is the standard object-command or noun-verb sequence of selection. Entities appearing on the screen are either acted upon or selected. Selecting entities is a primary goal in the user's conceptual model; then the user selects the action or change of state to be effected.

The Xerox Star was a pioneering achievement in the corporate identity of interface design. It represented a state-of-the-art, object-oriented screen manipulation in a high-resolution system. Of considerable importance to this discussion is the effort that was undertaken to design not only the algorithms that support the system but also the manner of its representation in a systematic form. The conventions established by the Star have already begun to influence later systems.

The Apple Lisa. Following the Star's approach is the Apple Lisa (see Figure 2), which appeared in January 1983.[13] Like the Star, the Lisa offers a selection of typefaces, including serif, sans-serif, fixed-width, and variable-width letters, and a half-dozen variations in type style including shadowed letters. To utilize too many of these typefaces in the interface itself would not represent the corporate identity approach; and the Lisa designers wisely chose to display primarily a single size of modern sans-serif letters in upper- and lowercase, with selected screen buttons appearing in reversed type.

Some of the Xerox Star's development staff came to work on the Lisa and influenced Apple's designers to adopt the Star's desktop concept as a unifying metaphor. Although the terms "iconic" and "representational" are relativistic terms for representational or abstract signs, in different systems one is able to distinguish specific differences in approach. The icons of the Lisa are in some cases more highly representational and detailed than in the Star, for example, in the garbage can icon that (somewhat

confusingly) represents a wastebasket for unneeded files, indentations appear along the sides of the can and even a handle is added on the top to raise the lid. These details suggest the beginning of more illustrative or anecdotal icons for more personalized workstations.

In the Lisa, the icons appear with their verbal equivalents directly below them. The windows themselves and the submenus that descend from the menu bar positioned across the top of the screen show slight drop shadows that begin to indicate an implied three-dimensional structure to the flat workspace of the desktop screen.

A typical difference in the interface style of the Star and Lisa systems can be seen in the stronger window-scrolling arrows used in both systems. In the Star the arrows are drawn with three thin lines; in the Lisa they are thicker, with an outline and a drop shadow. As mentioned before, the symbolism in the Lisa is richer and more representational from a visual communications point of view. The symbol set also begins to show some weakness as a completely designed system. For example, the reversed titles of windows have unique ornamental additional lines to their sides, an unnecessary deviation from a total corporate identity approach.

Color considerations in the Lisa are similar to those in the Star. The Lisa also uses several gray-value textures to distinguish the primary desktop, windows, and window borders, but the exact grays are slightly differently disposed in comparison with the Star. Of special note in the Lisa are the gray right side, gray bottom, and the white top border of the windows.

The Apple Lisa permits a relatively unorganized location of icons on the desktop. One strong gridded feature of its screen design, however, is the menu bar that appears at the top of the screen. Submenus "pop down" (rather than up) temporarily from it and may obscure material appearing below. As formulated by both the Star and the Lisa, the metaphor of the desktop does not include any corporate standard for desk organization or windows.

The noun-verb selection paradigm of the Star has been incorporated into the Lisa: The user selects objects first and then the transformations intended for those objects. As for the Star, the windows, icons, and submenus are intended to appear and disappear instantaneously; if this does not happen as intended, it is treated as a deficiency to be hidden with whatever means available. In the Lisa, the window that appears when an icon is pressed zooms up from the icon position in a noticeable transition. Many computer graphics display systems assume that faster is always better. However, a communication-oriented approach might suggest for novice or occasional users—especially during training periods—that dissolves, fades, wipes, and zooms might be of value in communicating the meaning of change.

The Apple Lisa system represents a substantial refinement of the ideas introduced in the Star system. In January 1984, Apple introduced the Macintosh.[14] This low-cost version of the Lisa continues many of its graphics features and makes evident the value of corporate graphic standards for interface design. (See box on page 30.)

The Intran Metaform system. The Metaform system (see Figures 3 and 4) from Intran represents a modest

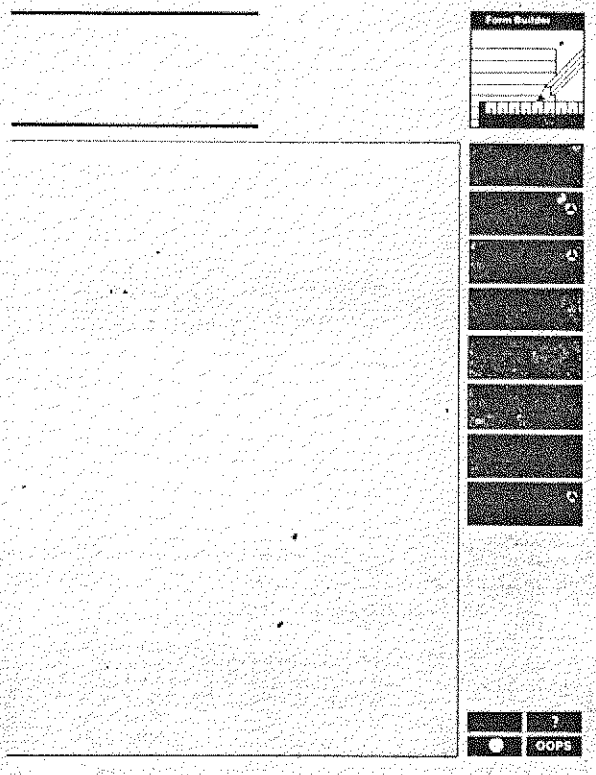


Figure 3. Prototypical screen layout, designed by Aaron Marcus and Associates, for the Intran Metaform system. Visible are iconic screen buttons, an illustrative cursor, and areas for menus and submenus.

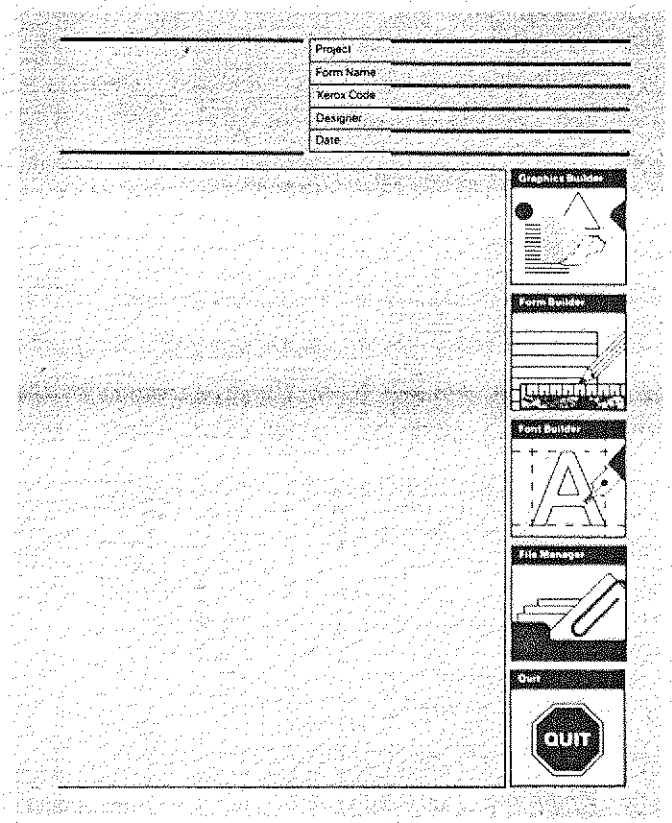


Figure 4. Another screen layout for the Intran Metaform system.

development scale in comparison with the 30 work years of development on the Star's interface and the 200 work years claimed for the development of the Apple Lisa.[15] Interface design for the Metaform system consumed only a few work years by the time the system was announced in December 1982. Metaform provides special-purpose software to accomplish forms design and editing for the Xerox 9700 laser printer. The software resides on a PERQ high-resolution display system.[16] The Metaform's system-oriented graphic designers prepared from the very beginning of the project for a corporate design approach when they proposed prototype sketches for typography and screen layout.

The design scheme called for most screen displays to accommodate Univers topography in a single size for all systems messages. Reversed type is used for selected buttons. Multiple lines of type always appear stacked flush left and ragged right, which makes the scanning of lists of buttons and other information easier.

Iconic symbolism for the Metaform light buttons took a special direction. The corporate graphics convention for primary module icons is to use narrative images that tell a brief story about what occurs within a module of the system. These large icons are somewhat like illuminated letters in medieval manuscripts. In comparison with the Star and the Lisa systems, Metaform's images are much more iconic. They contain elaborated detail related to activities and to other signs used in the system. They indicate appropriate cursors and explain to the uninitiated what will happen in the modules. Similar to the Star and the Lisa, Metaform uses different cursors to signal system processes and states to the user. Pen points, brushes, pencil points,

One approach to the human-computer interface

Torrey Byles, Contributing Editor

The underlying design philosophy of the Apple Macintosh has been that it will be easy to use—so easy, in fact, that users will become functionally familiar with the machine and most of its software in just hours. This ease-of-use philosophy, although rooted in a major company (Xerox), is a radical departure from traditional computer design and puts the Macintosh in a category of its own.

The key to the design is the way the computer interacts with the user in human terms instead of the usually cryptic computer terms. This is not just a matter of colloquial phrases in the menu selections; it means the computer communicates with the user by easily understood, nonlinguistic symbols and allows the user to respond by tactile movements that soon become second nature.

Central to this interactive mode is the Macintosh's mouse, a pointing device that allows the user to select commands or locations on the screen almost by pointing a finger. To make selections on other computers, the user would normally depress special function keys and directional cursor movement keys. The mouse, however, allows the user to make a menu selection by depressing a single button.

Many computers describe menu choices with obscure phraseology. The Macintosh describes them symbolically, with appropriate pictures, or icons. For example, by selecting a pair of scissors, the user can cut out a section of text or graphics to be used elsewhere; choosing the aerosol can turns the mouse into a can of spray paint for adding fine-mist shading to drawings.

Options for the various graphics and type styles, such as shadings, fonts, and polygonal forms, are pictorially displayed on the screen—again, without any language description. In fact, except for menus of single words that can be translated easily into all languages, the Macintosh uses no English language in or on the machine. Even its ROM contains no English code, according to its fabricators. Since the Macintosh is not language specific and utilizes a universal symbol system, it is easy to use for people of any culture (a benefit whose marketing merits are not unintentional).

Adding to the ease-of-use capability of the mouse and icons is the Macintosh's high-resolution (72 pixels per inch) display, which simulates the actual desktop working environment—complete with built-in notepads, file folders, calculator, and other office tools. Rather than light text on a dark background (as in most displays today), the Macintosh's nine-inch, 512 x 342-pixel screen mimics printed paper pages by reading out black type on a light gray background. This ergonomic feature reduces eye strain and reflections from other nearby light sources.

Integral to the desktop simulation are the windows, which can display several documents (text and/or graphics) simultaneously. They can also be moved, expanded, or shrunk. This means that numbers, words, and pictures can be "cut" from memos, charts, or graphics and "pasted" into other documents—even those created in separate application programs. An exploding window capability allows the user to scan menu options quickly, without going through the time-consuming process of re-creating the entire screen.

According to Apple Computer, third-party software vendors will employ the Macintosh's icon/window/menu features, thereby maintaining a standard, easy-to-use interaction capability.

Macintosh hardware is compact but powerful. The CPU/monitor assembly takes up about the same amount of desk space as a piece of paper, yet its single nine-inch-square circuit board hosts a 32-bit microprocessor with a CPU running speed of 7.83 MHz. Standard internal memory is 128K of RAM and 64K of ROM. Users have the option—at $995 extra—to purchase 512K of internal memory.

Secondary memory consists of one built-in 3½-inch disk drive with single-sided 400K storage capacity—100 pages of double-spaced text. In the back of the unit is a connector port for an additional disk drive.

The free-standing keyboard consists of 58 keys in a full-stroke, Selectric-style layout. The Macintosh's small size allows the user to move the machine easily on his desk and even to and from his work.

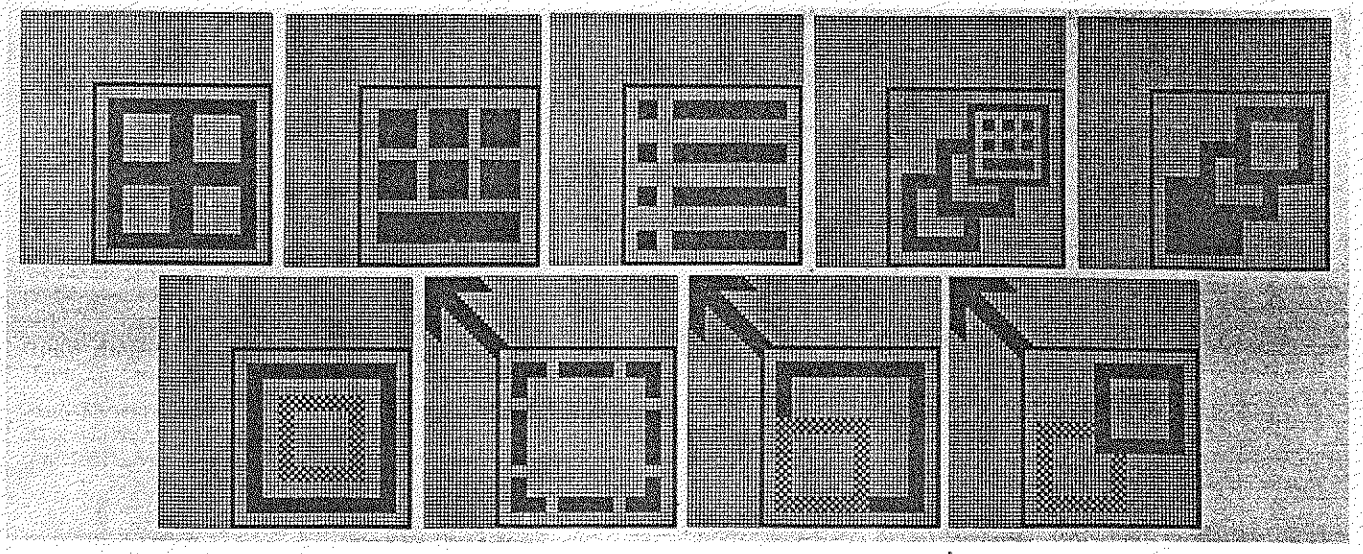


Figure 5. Examples of icon and cursor designs (by Aaron Marcus and Associates) for the PERQ Accent operating system. All elements of the signs have been carefully limited to a set of marks that can be combined into the different forms required for the functions of the system.

paper clips, and pointing hands all contribute to explaining and differentiating activities.

Color distinctions in Metaform are similar to those in the Star and the Lisa. Gray values are used to distinguish different areas such as light-button fields from the work area. Gray allows both white and black to be used for highlighting and low lighting.

In the original Metaform design (actual implementation differs slightly in some features), the screen is a tiled surface of areas. Unlike screen layouts that permit a variety of window areas to appear anywhere, Metaform was originally intended to emphasize the regular appearance of windows in fixed locations and sizes on the screen. This feature contributes to the user's knowledge of the system and reduces the spatial clutter associated with items appearing and disappearing.

The menu appears at the right of the screen, while the work area appears at the left. (This could easily be reversed for left-handed users.) By consistently locating screen components in one place, users can be helped to learn the layout of the system and its features more quickly. The forms designed within the work area can be of any organization, depending upon the particular function of the form.

With respect to sequencing, the Metaform design solved the problem of the appearance of submenus after a screen-function button had been selected by having the submenu buttons descend from the selected item much like a window shade, while other buttons slowly rearrange themselves to accommodate the new functions. (In addition selected buttons popped out *three-dimensionally* using a drop-shadow technique.) Besides the visual interest that such a small-scale animation sequence provides, the user can see a visual representation of menu hierarchy. This helps the novice or occasional user to comprehend the changes that take place in screen organization and system functionality. Clearly, for the experienced user the speed of this feature needs to be a variable parameter so that it can keep pace with the user's familiarity with the system.

From its very beginning Metaform designers intended that the system exhibit comprehensive, systematic graphic design in all aspects of the system: screen design, off-line user documentation, and marketing materials. Practical limitations of implementation precluded the extent to which corporate graphic design could shape the commercial form of the system; however, the approach and its achievements are clear. The Metaform system represents a significantly smaller investment of time and effort than the other systems but indicates that corporate design standards may be built effectively into a system with considerably smaller resources than those available for the development of the Star, the Lisa, or the Macintosh systems.

The Metaform approach to corporate identity is being carried forward in current graphic design for Accent, a programmer's operating system on the PERQ 2. See the article by Brad Myers in this issue of *IEEE Computer Graphics and Applications*. The more systematic approach is evident in examples of icon and cursor designs shown in Figure 5. These signs attempt to improve upon the original designs developed by Myers. They simplify and coordinate the variety of marks used to identify and to differentiate functions. They also attempt to equalize the visual weight of all the signs, to reduce visual clutter, and to utilize repetition of forms as an aid to the learning of a new visual code.

Future developments

Although iconic interface design is just beginning to be introduced into the commerical market, the field is expanding rapidly as new systems based on microprocessor technology and high-resolution display are developed. The computer graphics industry already has some strong, clear prototypes upon which to base new designs. Several important issues arise that must be explored in the next stages of graphically designed interfaces:

- What is an appropriate screen format: square, vertical rectangle, or horizontal rectangle? What proportions should a rectangular screen possess?
- Where should menus be located ideally: at the top, the right edge, the bottom, or should they be floating freely?
- Should window organization be free and unorganized, or should some default layouts be imposed to aid comprehension, memory, and user efficiency? What size and shape should windows have, and should they have borders?
- Should function buttons, objects, and other illustrative imagery be very representational or abstract?
- What is the ideal typeface for an interface: positive or reversed; serif or sans serif; one size or several sizes?
- Should screen elements appear and disappear quickly or slowly? How can dissolves, wipes, cuts, zooms, and other cinematic techniques be incorporated effectively?
- How can color be used effectively to enable users to learn more quickly and to be more efficient in performing their tasks?

The answers to many of these questions will emerge in the iconic interfaces that will be designed in the next few years. Some of these systems will help establish the conventions for the corporate design of systems in areas other than office automation, such as CAD/CAM and computer-aided learning.

Current developments present an exciting challenge to the computer system designer and the graphic designer who can and should work together to create effective interfaces for powerful computer graphics systems. As the field of high-resolution iconic interface design matures and adopts corporate design conventions and, eventually, standards, the entire community of builders and users will benefit by being able to see and learn from successful approaches to typography, symbolism, color, layout, and sequencing. ■

References

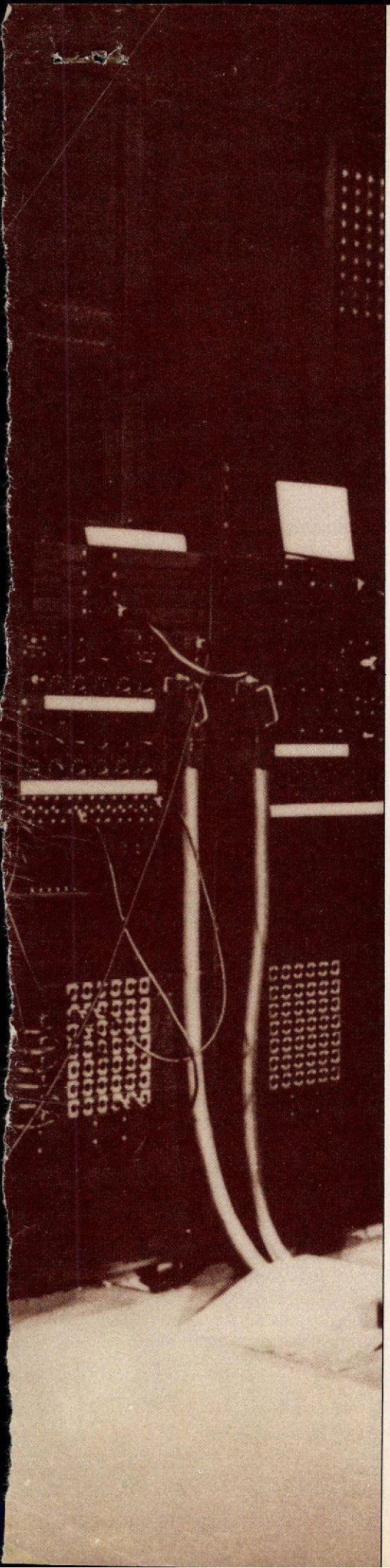
1. Aaron Marcus, "Graphic Design for Computer Graphics," *Proc. Intergraphics 83, Technical Session B5-2,* Tokyo, Japan, 1983, pp. 1-9; and *IEEE Computer Graphics and Applications,* Vol. 3, No. 4, July 1983, pp. 63-68.
2. James D. Foley and Andries van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass., 1982.
3. Aaron Marcus, "Designing the Face of An Interface," *IEEE Computer Graphics and Applications,* Vol. 2, No. 1, January 1982, pp. 23-29.
4. Davied Canfield Smith, et al., "Designing the Star User Interface," *Byte: The Small Systems Journal,* Vol. 7, No. 4, April 1982, pp. 242-282.
5. Ann Chaparos, *Notes for a Federal Graphic Design Manual,* Chaparos Productions, Washington, DC 20001, 1979.
6. Rolf F. Rehe, *Typography: How to Make it Most Legible,* Design Research International, Carmel, Ind., 1974.
7. Josef Mueller-Brockman, *Grid Systems in Graphic Design,* Verlag Arthur Niggli, Niederteufen, West Germany, 1981.
8. Umberto Eco, *A Theory of Semiotics,* Indiana University Press, Bloomington, 1976.
9. Aaron Marcus, "An Introduction to the Visual Syntax of Concrete Poetry," *Visible Language,* Vol. 8, No. 4, autumn 1974, pp. 333-360.
10. Otl Aicher and Martin Krampen, *Zeichensysteme der visuellen Kommunikation,* Verlagsanstalt Alexander Koch, Stuttgart, West Germany, 1977.
11. *Symbol Signs,* American Institue of Graphic Arts, Visual Communication Books, Hastings House, New York, 1981.
12. Aaron Marcus, "Color: A Tool for Computer Graphics Communication," in *The Computer Image,* Greenberg, D. et al., eds., Addison-Wesley, Reading, Mass., 1982, pp. 76-90.
13. Gregg Williams, "The Lisa Computer System," *Byte: The Small Systems Journal,* Vol. 8, No. 2, February 1983, pp. 33-50.
14. Gregg Williams, "The Apple MacIntosh Computer," *Byte: The Small Systems Journal,* Vol. 9, No. 2, February 1984, pp. 30-40ff.
15. "Metaform User's Manual," Intran Corporation, Intran Image Management Group, 7429 Bush Lake Rd., Edina, Minn. 55435, 1984.
16. *Perq 2 User's Manual,* PERQ Systems Corporation, Pittsburgh, Penn., 1984.

Aaron Marcus is the principal of Aaron Marcus and Associates, Berkeley, California, an information-oriented graphic design firm, specializing in effective communication for computer graphics display. Previously he was a staff scientist in the Computer Science and Mathematics Department of Lawrence Berkeley Laboratory. From 1968-1977 he taught at Princeton University and was a consultant in computer graphics at Bell Telephone Labs, Murray Hill, New Jersey, where he programmed a prototype interactive page design system. His professional graphic design work and computer graphics have been exhibited, published, and awarded internationally. He has written and lectured extensively on graphic design and on computer graphics for professional journals and conferences of both fields.

Recently, Marcus has coauthored *The Computer Image* published by Addison-Wesley; written *Managing Facts and Concepts* published by the Design Arts Program, National Endowment for the Arts; and written *Soft Where, Inc.*, Vols. 1 and 2, published by the West Coast Poetry Review. He is on the advisory boards of *Computer Graphics Today* and *Information Design Journal.* He has consulted internationally with major computer graphics research and development groups on the subjects of chart, diagram, and map design; on user-friendly iconic interfaces; and on program visualization/documentation techniques.

Marcus received the BA in physics from Princeton University and the BFA and MFA in graphic design from Yale University Art School. He is a member of the NCGA, ACM, Siggraph, Yiem Computer Arts Society, Sigma Xi Scientific Organization, and the American Institute of Graphic Arts.

The author may be contacted at Aaron Marcus and Associates, 1196 Euclid Avenue, Berkeley, CA 94708-1640.



Don't fall behind with an obsolete system

It's more than sad when a system is too old to work properly—it's costly. Here's how to recognize, and avoid, that sorry state.

by David Kull, Management Issues Editor

Obsolescence, always a creeping phenomenon, creeps more rapidly these days. Technological developments accelerate.

Major new computer lines appeared about every eight years during the 1960s and 1970s; that cycle spins almost twice as fast now. Other aspects of information technology advance just as rapidly. Meanwhile, demands on information-resources departments multiply as businesses turn to them more and more frequently for a competitive edge. As systems age more quickly, determining when they are simply too old becomes more difficult—and more important.

Two forces push systems toward obsolescence—economics and functionality. A system is ready for replacement when the costs of keeping it exceed those of changing to another, or when it cannot meet the corporation's needs. Often, these forces work in tandem, making the decision to change a judgment that combines the desire for improved services with a need for cost-effectiveness.

Figuring a system's operating costs is simple mathematics. The key is to include *all* costs. Lease payments and depreciation are only the start. There are also expenses for space, cooling, and power. The newer generations of smaller, power-efficient cpus provide considerable savings in these valuable commodities, sometimes making room and board for the older models unacceptably expensive. And costs for technical support, which are likely to increase as a system approaches retirement, can become exorbitant—particularly when a vendor discontinues a line. These "ancillary" costs would make some systems uneconomical even if you could get the systems for free.

You must also tote up expenses for a new system. Determining them requires careful projections, particularly when the new technology is only a gleam in the manufacturer's eye. You must monitor the choices that are available or soon will be, however. An old system becomes obsolete as soon as superior alternatives appear.

According to Robert V. Head, president of CAPIT (Company for Analysis and Planning of Information Technology) in Stafford, VA, there are a number of industry observers who can project the trends in cost and performance for

Photo courtesy of Sperry Corporation

Obsolete Systems

(Continued from page 165)

mainframes about five years ahead with a high degree of confidence. Most mainframe vendors, with the important exception of IBM, will advise customers on developments if they sign non-disclosure agreements. Head recommends engaging a technology-forecasting service, such as the Gartner Group or Yankee Group, to help you keep up with Big Blue.

When considering a spanking-new system, you must give thought to the probability of *its* obsolescence. Vendors recognize the shortened life cycle of systems and have tightened leasing arrangements accordingly. By charging a premium for shorter-term leases—of three or four years, as opposed to the traditional seven—or by jacking up the bail-out penalties, they're assured of turning profits by the time the customer starts looking toward the next computer generation. The primary recourse, besides being aware of the potential problem, according to Robert Head, is to shop for the best deal. For most corporations, this means checking out the plug-compatible-mainframe manufacturers.

Monitor available options. An old system becomes obsolete as soon as there are superior alternatives.

According to Kailash Khanna, vice president for strategic systems planning at American Express Co. in New York, organizations can retain considerable flexibility in meeting their mainframe needs. Vendors, including IBM, describe their products about a year before bringing them to market. "If you're looking a year or two ahead, you can plan to use what you know is available or what you expect to be available," Kailash points out. "Then, when the time comes, you can take the best course."

Organizations running at least two mainframes can halve the risk of being stuck with an obsolete one by staggering the end dates of the leases. F. William Hoffman, a consultant with Price Waterhouse in New Orleans, points out that "leapfrogging" two eight-year leases is almost as good as having four-year commitments for each machine. You can change one unit every four years as your workload demands.

In assessing the risks of signing a lease, a company should consider the length of time the product line has been on the market. The younger the model, the less risk in making a long-term commitment. On the other hand, even a four-year lease would be imprudent if the model has been on the market six or seven years and is about to be replaced. Hoffman tells of a steel manufacturer that leased two IBM 360 series processors fairly late in the line's life cycle. Even with moderate workload growth, Hoffman says, the equipment was inadequate—and technically obsolete—long before the lease expired.

"Don't think you can outsmart the lessors," warns Hoffman. "If they're offering equipment at a very low cost, it's for a reason."

Hoffman believes reliable, realistic capacity planning is the key to avoiding obsolescence. An organization should review these plans annually to be sure that the projections hold true. Companies make some common mistakes in predicting their horsepower needs. Many times, they will underestimate the transaction volumes for new online systems or the demands of sophisticated databases and high-level programming languages. Professionals forget that online pro-

Assessing the situation

Data-processing managers in American Can Co.'s metal-packaging division recently compiled reports in the format below to assess their 32 major application systems. (The form shown describes the kind of information to be gathered, rather than presenting a sample report.) The managers passed the reports to top management, recommending replacements of six systems.

System Profile

A. System Name

1. Primary Function:	Briefly describe the system's primary objective.
Primary User:	Identify specific departments using the system—the "owners" of the data and reports.
2. Secondary Function:	Where applicable, the system's secondary uses. An accounts-payable system's primary function is to maintain the company's payable liability and to generate payments to vendors. A secondary use is to provide information for tax reporting.
Secondary User:	For an accounts-payable system, a secondary user would be the corporation's tax department.
3. System Age:	How long has the system been in use?
4. Business Supported:	Which business units or divisions use the system?

B. Design Characteristics

Describe the system in the user's terms, emphasizing business, rather than technical, characteristics. Give processing frequencies (daily,

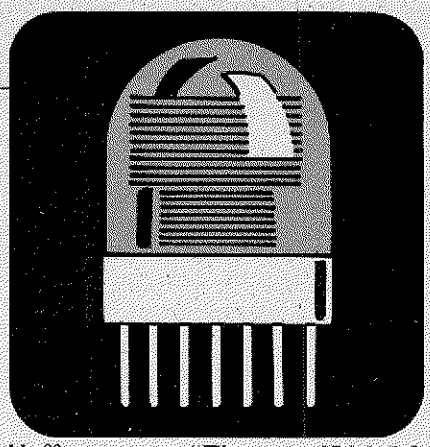
cessing, unlike batch operations, can't be transferred to after-hours or weekends.

"An organization will have stable capacity needs or gentle upward trends for several years," Hoffman says. "Then it will put in a sophisticated new system and be slow to adjust to the new demands. Some organizations end up adding or converting to a new machine every year."

Companies expecting increases in processing needs can leave themselves a margin for error by acquiring a computer near the bottom of a line of compatible machines, Hoffman notes. If necessary, they can renegotiate the agreement and migrate upward fairly painlessly. But those anticipating only moderate increases in demand might dig themselves into a hole by signing a long-term lease for a top-of-the-line model.

A development that may complicate capacity planning for almost all dp shops is the personal-computer boom. Personal computers can push mainframe needs up or down. Some experts see the possibility of downloading applications from the mainframe. Others see vast numbers of personal computers, functioning as terminals, pumping data into the mainframes and doing some processing there. It's too soon to know the strength of either effect in an organization. While observers disagree over the degree of impact the personal computer will have, they all acknowledge that it injects at least some additional uncertainty into the planning process.

"Personal computers are likely to increase capacity requirements,"



Hoffman says. "The new IBM PC 3270 makes it easy for users to make demands on the mainframe, for example. But at this point, the impact is not strong enough to dramatically affect projections."

Robert Head sees more volatility. "Most assumptions about capacity have to be thrown out," he argues. "A bank with a 2 percent increase in account-activity rates every year for the past 15, for example, can no longer extrapolate that trend."

It's not enough to keep abreast of changes in million-instructions-per-second rates and storage costs. Advances in peripherals or communications can render an entire system obsolete. This applies to disk devices, printers, and other specialized add-ons. In the 1970s, for example, a new generation of check-sorters made that chore much easier—for those banks with mainframes that could accommodate the required operating system. Others had to switch central processors to keep up.

A comprehensive assessment of current system capabilities and future needs can cost from $10,000 to $100,000 in consultants' fees, according to Robert Head. Such a study should project about five years into the future, with fairly firm projections covering the first two years. Annual reviews need be only thorough enough to ensure the assessment's continued validity. After a full study establishes a baseline for an entire system, a company might break down its annual reviews, focusing on a particular aspect of technology each year. It might take a close look at the operating system one year, mass storage the next, and so on.

It's not a good idea to have leases for peripherals fall due at times other than when the mainframe commit-

weekly, monthly, etc.) and pertinent processing statistics (number of invoices per month, checks per month, etc.).

C. Annual Operating Costs

Include computer-operations costs, and data-entry, data-control, and telecommunications charges, as well as any equipment cost for those units that were acquired for specific applications (special terminals, printers, etc.). Also include any systems and programming charges for maintenance and development activities. Report these costs individually and, where possible, differentiate between fixed, allocated costs and direct, out-of-pocket costs.

D. Annual Non-dp Personnel Costs

Report costs for personnel who have been specifically hired to support the application system and spend more than half their time in that capacity.

E. Functional/Technical Comments

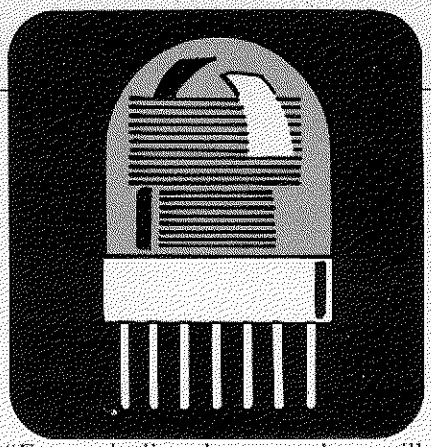
Comment on significant operational characteristics that affect the value or use of the system from the user's viewpoint. Comments may deal with the amount of manual intervention or checking required, the ease with which the user can interact with the data, any design attributes that preclude desired activities, or any business opportunities that may not be possible because of the way the system operates. Also, discuss the system's technical limitations (in nontechnical terms), and clarify coding and file-structure complexities and how they may inhibit quick, efficient data modification. (You may, for example, point out the lack of online processing efficiencies, such as editing and data validation.)

F. Recommendations

Suggest both short- and long-term actions. Generally, the recommendations will fall into one of three categories: 1) Continue using the system with normal maintenance; 2) Make significant modifications or upgrades; or 3) Replace the system. Conclude with estimates of costs, paybacks, and potential risks.

Obsolete Systems

(Continued from page 167)



ments expire, however. An upgrade in one piece of equipment might make another obsolete. If the lease on the second still has years to run, you may pay a heavy price for that obsolescence.

An organization's tax specialists should review every lease or purchase agreement *before* it's signed—but the accountants should not do their part without the advice of the technical staff. William Hoffman tells of an electric utility whose accountants assumed a new mainframe had a 20-year life—and depreciated it at that rate for tax purposes. When the dp department found it needed a new system after only four years, its managers figured the utility had at least written off a large share of the old machine's value. The fact that it had not didn't necessarily cost extra—the full value would be written off eventually—but management's dismay at the accounting loss did not work in data processing's favor.

Companies should not rely too heavily on economic analyses in making their acquisition decisions. These analyses always point to the same conclusion, Hoffman contends. "Economically, a long-term lease will always look better than a short-term one," he says, "and if the equipment's useful life is long enough, it's always cheaper to purchase."

Technical considerations, however, often turn those seemingly sure gains into bad bets. Hoffman describes a cash-rich bank that took advantage of a good purchase price on an IBM 370/155. The 370/158 with virtual storage came onto the market a few months later, slashing the value of the bank's machine. In fact, the bank had to buy a $200,000 conversion unit just to keep up with the new technology.

Systems can become outdated in many ways, not all of which would be obvious to the nontechnical professional. Sometimes, even the dp staff won't recognize the problem's extent until it's too late. This happens when an application dies a slow death from too much patching. Rather than revise a system in an orderly fashion as user needs change, data processing applies "temporary" fixes. As these patches add up, service declines and maintenance costs soar. Eventually, a major system overhaul will be necessary. And it will be more difficult and costly than it would have been without the awkward postponements.

"The way to avoid such a predicament," advises Kailash Khanna, "is by keeping in touch with users about their needs."

Usually, an organization's dp needs evolve, but occasionally they change suddenly. American Can Co. in Greenwich, CT, for example, recently began a diversification into financial services and other nonpackaging businesses. At about the same time, it began decentralizing its data processing.

It was obvious that the metal-packaging division, which had some of the oldest dp equipment in the company, would have to respond to the new situation, according to Joseph C. Donia, its managing di-

Playing catch-up in the public sector

How old are the information systems that serve the U.S. government? So old that manufacturers have stopped supporting more than half the equipment that operates them. So obsolete that by upgrading them, the country could gain more than $29.5 billion in savings and revenue enhancements over the next three years.

These are among the findings of the President's Private Sector Survey on Cost Control. The recent report, by the task force of business leaders that studied the government's data processing and office automation, identified general and specific problems.

One of the basic problems is simple age. The average age of the government's 17,000 larger computers is almost seven years, the task force says. (In private industry, the average age of automated data-processing equipment is under three years.) One reason the systems are so old is that government agencies must go through a lengthy acquisition process. Requirements instituted in the 1960s to control the proliferation of government computers and ensure competition for bids often cause considerable delays. The task force cites a U.S. Forest Service attempt to award a system contract in which seven reviews of the agency's request for proposals added a year to the acquisition cycle.

The report does not pin all the blame for the obsolescence of government systems on these procurement controls, however. The task force also sees a deficiency in the government's information-resources leadership.

"The government has failed to develop a coherent system for [data-processing] planning and management," the report states. "As a result, it has not capitalized on the substantial opportunities for cost savings and effectiveness improvement."

Among the task force's recommendations is the appointment of a federal information-resource manager to oversee a government-wide steering committee. It also recommends that each federal agency hire full-time, professional information-resource managers. Most agencies now assign responsibility for the information function to an undersecretary who has several other duties as well.

Obsolete Systems

(Continued from page 168)

rector of information systems and services. But just how much would it have to change? And in what direction would it have to move?

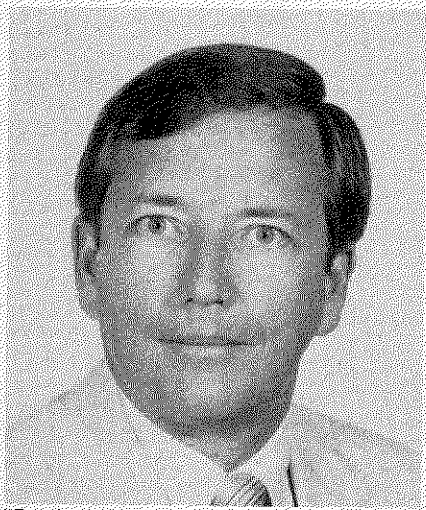
To find the answers, Donia developed a form on which his six dp managers could profile the 32 systems for which they were responsible. Each system represented a major application, such as accounts payable, accounts receivable, and payroll.

The forms (See the box on Page 166.) provided spaces for the managers to describe each system's functions, technical characteristics, and costs. They also allowed for comments and recommendations. Since the reports would be directed to general management, the managers used nontechnical language as much as possible. Rather than discuss I/O (input/output) statistics, for example, they wrote about the number of checks processed. The data center provided most of the descriptive information. The managers also talked with end users about their needs and concerns before writing the comments and recommendations.

The system-review form provided for three general recommendations: Continue the system with normal maintenance, significantly modify or upgrade it, or replace it. Each recommendation was supported by a discussion of the related costs, expected paybacks, and risks. The managers concluded that six of the 32 systems required replacement.

"Generally, these systems were not designed for a decentralized operation; they weren't responding to users' needs," Donia says.

"When we looked at our systems, some users were surprised at how much manual effort was going into automated tasks."
Donia, American Can



"Don't think you can outsmart the lessors. If they're offering equipment at a very low cost, it's for a reason."
Hoffman, Price Waterhouse

Donia sent the reports to his division's top management and to the corporate dp group. Their responses are still pending, but Donia is optimistic about gaining their support. The users' input will help. "When we looked at these systems, some of the users were surprised at how much manual effort was going into automated tasks," he notes.

Information-resource managers should also listen to staff to determine how up-to-date their systems are. A company should not upgrade its machines just to please programmers, of course. But it should be aware of the difficulty it will encounter in attracting and keeping people without having the latest technical enticements. William Hoffman

A development that may complicate capacity planning for almost all dp shops is the personal-computer boom.

points to high turnover among professionals as one of the hidden costs of obsolescence.

You should also keep an eye on the competition—competitors' advances may force you to play catch-up. Most airlines are still trying to close the competitive gap American and United opened in the late 1970s with their online reservations systems, for example.

If you *are* to miscalculate in figuring your organization's system needs, particularly its capacity requirements, it's best to err on the side of having too much power. Any loss from overinvesting in information resources should be minimal and will probably be recovered eventually as demand catches up. Underestimates can lead to losses that may never be recouped.

"If you don't have up-to-date systems, you may miss out on opportunities," says Kailash Khanna. "But the biggest hits come when equipment can't cope with demand—particularly in online systems. When that happens, you may actually be losing revenue." □

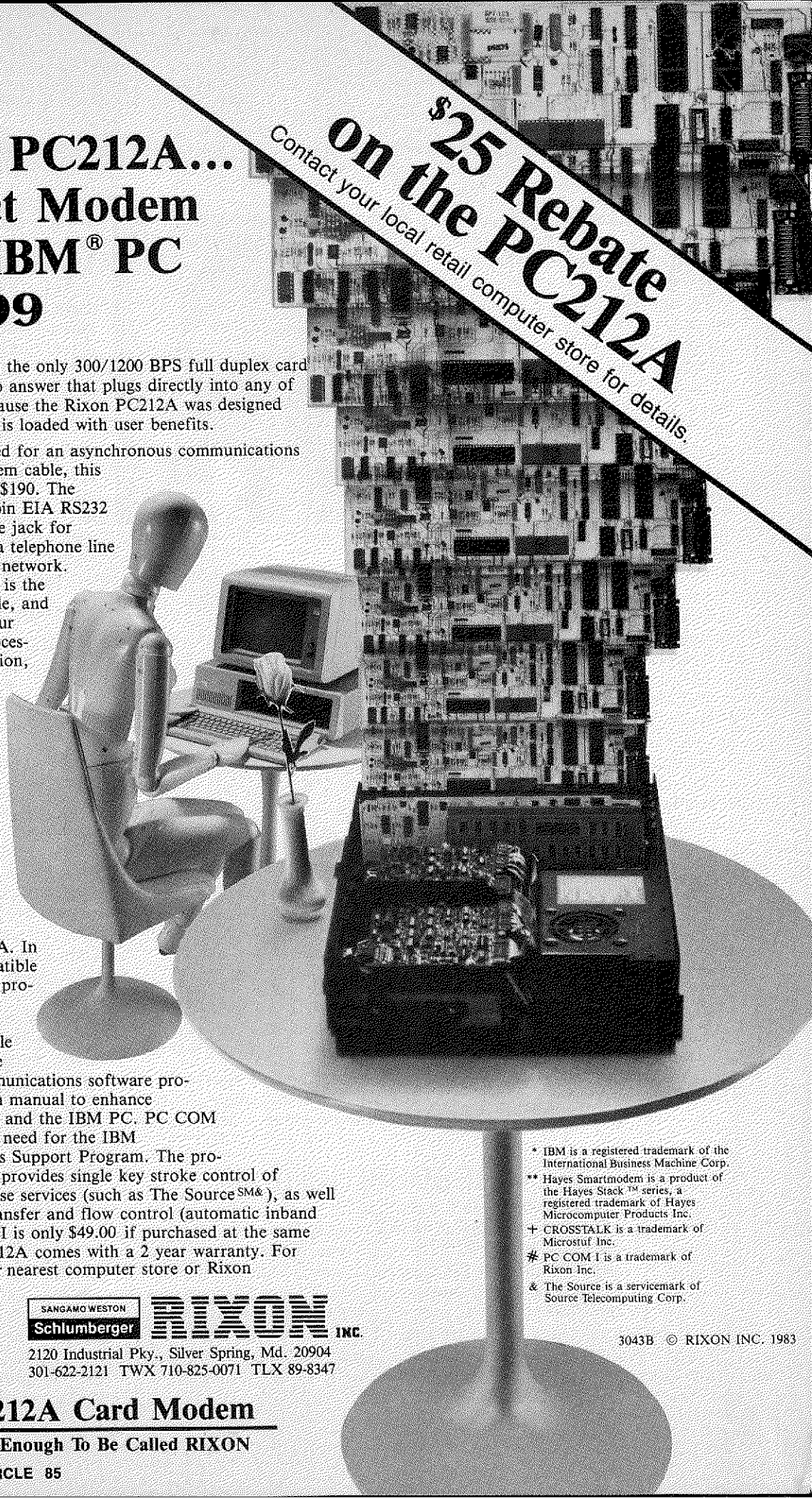
Automated assessment

Capacity planning can be a complex and formidable chore. But don't be fainthearted; help is available—in the form of systems that measure systems. Copernicus Model Release 2, for one, from Technotronic Inc. in McLean, VA, predicts the effects of hardware upgrades, workload increases, and new applications on cpu needs. The model analyzes IBM MVS and VM systems and presents graphic results. It's available for $7,000 as an option on the Copernicus 820 system for computer and network-performance measurement. For more information, call (703) 356-2151. *Circle 547*

The Rixon PC212A... The Perfect Modem For Your IBM® PC ...Only $499

$25 Rebate on the PC212A

Contact your local retail computer store for details.



The Rixon® PC212A offers you the only 300/1200 BPS full duplex card modem with auto dial and auto answer that plugs directly into any of the IBM PC® * card slots. Because the Rixon PC212A was designed specifically for the IBM PC, it is loaded with user benefits.

The PC212A eliminates the need for an asynchronous communications adapter card and external modem cable, this alone saves you approximately $190. The PC212A provides an extra 25 pin EIA RS232 interface connector, a telephone jack for alternate voice operation, and a telephone line jack for connection to the dial network. Without question, the PC212A is the most user friendly, most reliable, and best performing modem for your IBM PC. An internal microprocessor allows total control, operation, and optioning of the PC212A from the keyboard. A user friendly HELP list of all interactive commands is stored in modem memory for instant screen display. Just a few of the internal features are auto/manual dialing from the keyboard, auto dial the next number if the first number is busy and instant redial once or until answered. In the event of power disruption a battery back-up protects all memory in the PC212A. In addition, the PC212A is compatible with all of the communication programs written for the Hayes Smartmodem™ **such as CROSSTALK.™+Also available for use with the PC212A is the Rixon PC COM I,™ # a communications software program (Diskette) and instruction manual to enhance the capabilities of the PC212A and the IBM PC. PC COM I operates with or replaces the need for the IBM Asynchronous Communications Support Program. The program is very user friendly and provides single key stroke control of auto log on to multiple database services (such as The Source℠ &), as well as log to printer, log to file transfer and flow control (automatic inband or manual control). PC COM I is only $49.00 if purchased at the same time as the PC212A. The PC212A comes with a 2 year warranty. For more information contact your nearest computer store or Rixon direct at 800-368-2773 and ask for Jon Wilson at Ext. 472.

PC212A ..... $499.
PC212A WITH ASYNCH PORT ..... $539.

SANGAMO WESTON Schlumberger RIXON INC.
2120 Industrial Pky., Silver Spring, Md. 20904
301-622-2121 TWX 710-825-0071 TLX 89-8347

* IBM is a registered trademark of the International Business Machine Corp.
** Hayes Smartmodem is a product of the Hayes Stack ™ series, a registered trademark of Hayes Microcomputer Products Inc.
\+ CROSSTALK is a trademark of Microstuf Inc.
\# PC COM I is a trademark of Rixon Inc.
& The Source is a servicemark of Source Telecomputing Corp.

3043B © RIXON INC. 1983

The Rixon PC212A Card Modem

Another Modem Good Enough To Be Called RIXON

CIRCLE 85

PROTOTYPING: SHORTCUT TO APPLICATIONS

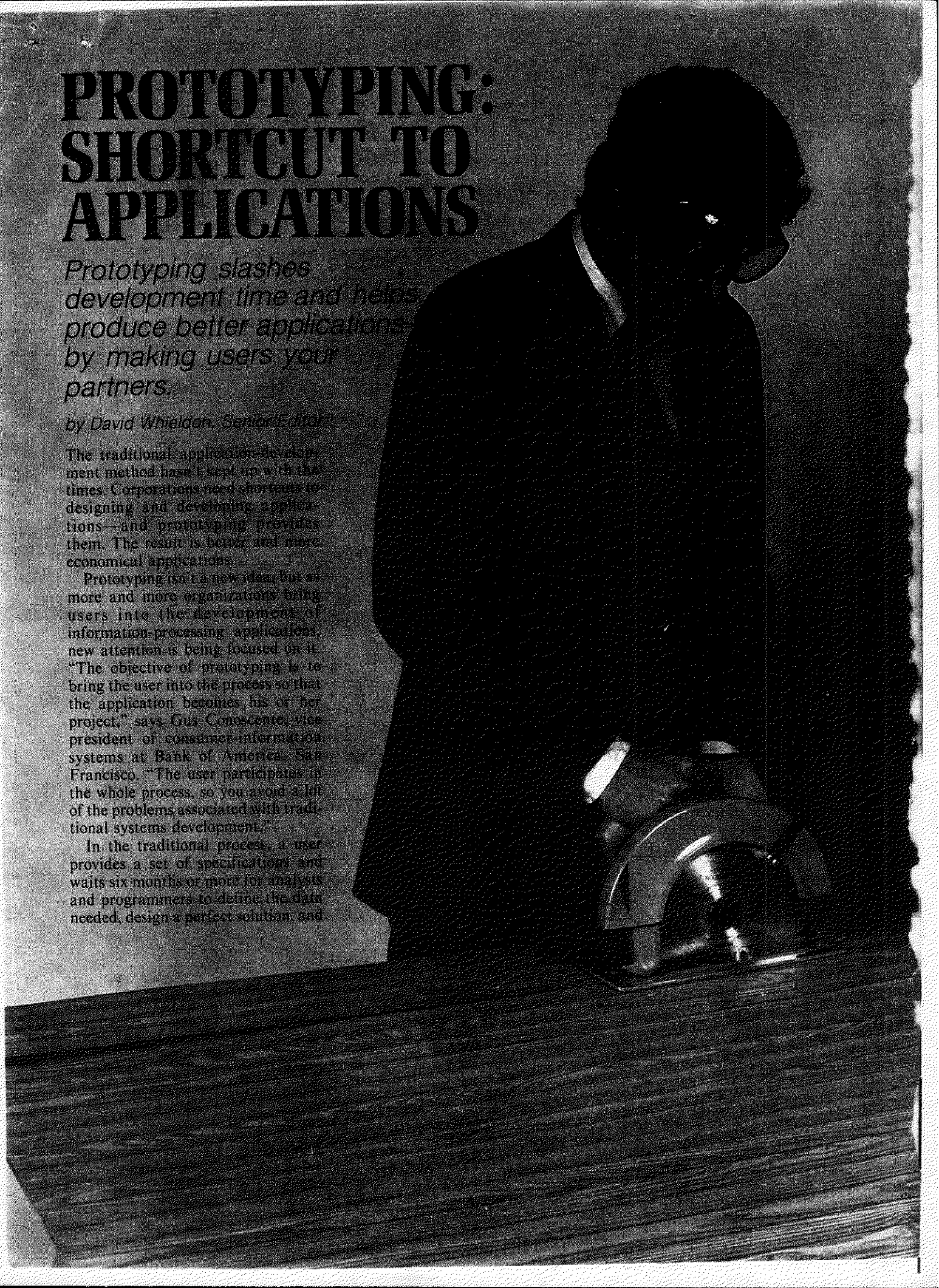
Prototyping slashes development time and helps produce better applications by making users your partners.

by David Whieldon, Senior Editor

The traditional application-development method hasn't kept up with the times. Corporations need shortcuts to designing and developing applications—and prototyping provides them. The result is better and more economical applications.

Prototyping isn't a new idea, but as more and more organizations bring users into the development of information-processing applications, new attention is being focused on it. "The objective of prototyping is to bring the user into the process so that the application becomes his or her project," says Gus Conoscente, vice president of consumer information systems at Bank of America, San Francisco. "The user participates in the whole process, so you avoid a lot of the problems associated with traditional systems development."

In the traditional process, a user provides a set of specifications and waits six months or more for analysts and programmers to define the data needed, design a perfect solution, and



Now Local Multiplexing is as Easy as Plugging in a Lamp

Within minutes you can put Line Miser™ multiplexers to work handling your local data traffic. Line Misers allow you to network your terminals, word processors, PC's and other data terminal equipment with minimal cabling requirements. The line savings can be tremendous! And now there are three types to choose from.

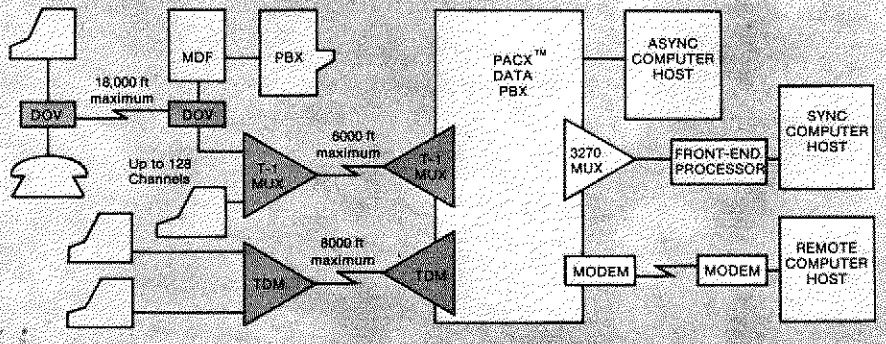
The popular Line Miser DOVs can turn your ordinary phone system into a versatile local area network supporting simultaneous data and voice communications. Everywhere you have a telephone you can quickly and easily add a terminal.

The new Line Miser GLM 528 combines T-1 speeds with large capacity. You get 128 async channels over a 1.544 Mbps T-1 link.

And for low cost local multiplexing you can't beat the new Line Miser GLM 510. In less than 3 minutes, you can add the GLM 510 to your private wire network to handle up to 8 async channels at 9600bps.

Bright ideas in local multiplexing. Three more reasons to switch to Gandalf. Ask your local Gandalf Sales representative for details today.

Line Misers™ make networks easier to build



gandalf™

Fully supported technology from concept to customer.

USA (312) 541-6060
Canada (613) 226-6500
U.K. Padgate (0925) 818484
Switzerland (022) 98-96-35

CIRCLE 61

(Continued from page 139)

workable. "If the information is laid out in vertical columns on a screen, the user might ask, 'What if we put it in horizontal rows?'" says R.F. Bellaver, division manager in application-development technology at AT&T Communications, New York. "In many cases, the prototyper can change the setup right on the screen."

The last phase is the key to the prototyping process, says Gary Guttman, president of Generation Science Inc., a contract-programming house in Syosset, NY. Building a prototype for users to play with *before* the application is put into final production heads off many potential problems, from bugs to inconveniences, and makes for a more flexible solution. "In traditional development setups, dp pros are constantly telling users to freeze the specs," Guttman says. "That's like saying, 'Freeze the world.' Requirements and conditions change, and systems have to be responsive."

It's during the experimentation stage that analysts and users probably will find critical elements they've overlooked in the application—"holes," in Guttman's lexicon. For example, Guttman's staff discovered a flaw in a security routine that would have prevented authorized users from accessing data and opened the door to unauthorized users. "We closed that loophole before the system went live," he says. "Probably the only way we would have found the problem was by prototyping."

Gaining a competitive edge

Fostering better relations with users won't be directly translated into an improvement of the bottom line. Nevertheless, prototypes may help to improve a corporation's ability to compete by reducing the amount of time needed to generate vital new information systems. As Higbee's, a Cleveland-based retailer, discovered, such benefits help produce concrete improvements of the balance sheet.

Like many other department stores, Higbee's has a bridal registry that helps the friends and relatives of brides-to-be choose appropriate wedding gifts. The registry was kept in a book, and it was updated—in theory—as purchases were made. However, says Pat McIntyre, vice president of management-information services, the theory was not the reality. "We were always subject to duplication and running two, three, or four days behind," he recalls. In addition, sharing information with Higbee's nine suburban stores was a chore. When top management heard last summer that an Akron-based affiliate of May Department Stores Co. (St. Louis) and the local outlet for J.C. Penney Co. (New York), its main competitors, were planning to automate their bridal registries, it issued a directive to beat them to it.

Working with a prototype, programmers and users can sort out an application's problems before making a big commitment, says F. Warren McFarlan of Harvard.

"If you could put a price on user satisfaction, the payoff of prototyping would be even higher than it already is." *Gaites, Westinghouse*

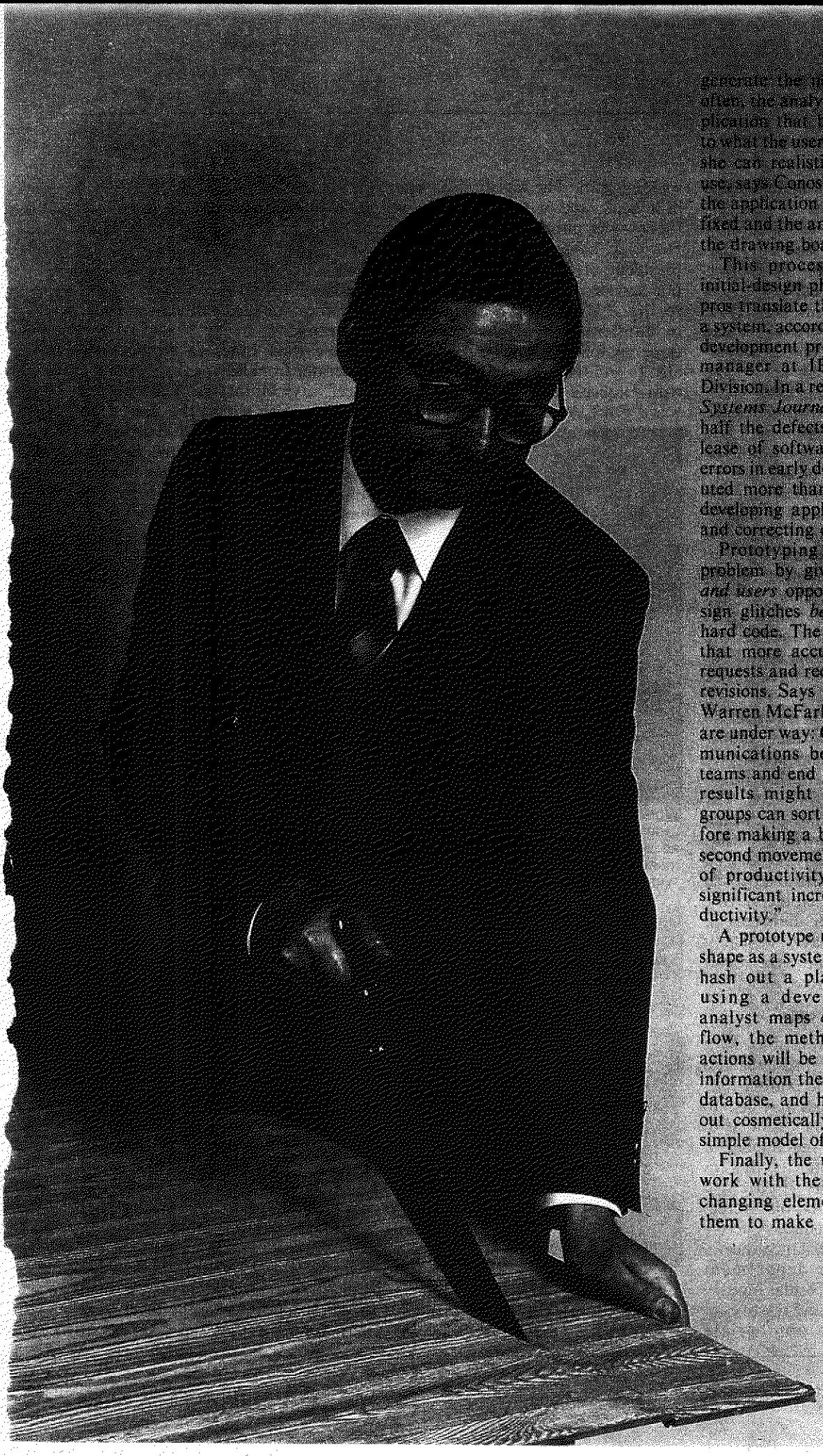
"We used Linc, an application-generator from Burroughs, to win the race," says McIntyre. "The software allows you to get to the user a lot faster than conventional methods."

McIntyre's staff started by producing a preliminary design for the automated bridal registry. Next, key employees were asked for their opinions and suggestions. "Our project manager showed the employees real, live screens to get the kind of feedback you don't always get with a blackboard discussion," recalls McIntyre. "We headed off problems we wouldn't have known about until all the work had been done."

The first prototype was completed last August, and four months later, Higbee's introduced its bridal registry—first in its market. The on-line system, based on a Burroughs B 1955, is now installed in 10 of Higbee's 11 stores. Each store's terminal displays each prospective bride's preferences in china, silver, and crystal as well as the items that have already been purchased. A bride-to-be's file is updated each time a purchase is made.

As for the payoff, McIntyre says the jury is still out. "Registrations appear to be higher this year," he says. "But we'll have to run a full year before we can measure the outcome." If the experiences of retailers around the country are an indication, Higbee's should see an increase in wedding-gift sales. Automating a bridal registry typically boosts bridal revenues by 20 percent, he says.

Meanwhile, McIntyre's staff is considering other applications that are appropriate for prototyping with Linc. "We've put in an advertising-scheduling and -control system using Linc," reports McIntyre. The system had been a priority for the advertising staff for some time, he says, but it



generate the necessary coding. Too often, the analyst returns with an application that bears no resemblance to what the user wanted or what he or she can realistically be expected to use, says Canoscente. In many cases, the application is too fouled up to be fixed and the analyst ends up back at the drawing board for a fresh start.

This process' weakness is the initial-design phase, in which the dp pros translate the user's request into a system, according to W. E. Beregi, a development programmer and group manager at IBM's Data Systems Division. In a recent issue of the *IBM Systems Journal*, Beregi wrote that half the defects found after the release of software can be traced to errors in early designs. He also attributed more than half of the cost of developing applications to detecting and correcting design flaws.

Prototyping attacks the design problem by giving system analysts *and users* opportunities to catch design glitches *before* they are set in hard code. The result is applications that more accurately reflect users' requests and require fewer expensive revisions. Says Harvard professor F. Warren McFarlan: "Two movements are under way. One is improved communications between development teams and end users about what the results might be. Together, these groups can sort out the problems before making a big commitment. The second movement is the development of productivity aids that promise significant increases in overall productivity."

A prototype usually begins taking shape as a systems analyst and a user hash out a plan on paper. Next, using a development tool, the analyst maps out the information flow, the method by which transactions will be handled, the kind of information the user wants from the database, and how the data are laid out cosmetically on the screen in a simple model of the final system.

Finally, the user and the analyst work with the screens, moving or changing elements or highlighting them to make the application more

Photo by Bill Oliwa

Prototyping

(Continued from page 140)

wasn't a corporate priority. Without the prototyping tool, the system probably would never have been completed.

Prototypes are usually constructed with tools that are designed for broader tasks. Higbee's, for example, used an application generator to build prototypes. Other software packages commonly called "programmer-productivity tools" are also appropriate for the task, including fourth-generation languages, documentation generators, and system-development tools.

Use.It from Higher Order Software is another tool used to build prototypes. Its Resource Allocation Tool permits simulation, making rapid construction of prototypes possible. Also, it contains a component that speeds data analysis and integration of routines.

Other products operate on personal computers. Excelerator from Index Technology Corp. runs on the IBM Personal Computer XT (the hard-disk version). It generates data-flow diagrams, structure charts, and data-model diagrams; paints screens; extracts reports from databases; and helps prepare documentation. Tools like Excelerator produce the best results if large systems are divided into small sections.

Ten times more work

Producing applications on time and to the satisfaction of users is a worthy benefit, but prototyping can also save money. Prototyping can directly cut the cost of producing applications by eliminating the fix-it coding common to traditionally developed systems. In addition, it can dramatically raise the productivity of data-center staffers, indirectly saving money. One of his staffers is doing eight to 10 times as much work using prototypes as he did developing applications via the traditional method, says Ed Gaites, director of broadcasting information at Westinghouse Broadcast & Cable Inc., Milford, CT.

Westinghouse used a prototype to set up a system to keep track of about

50,000 videotapes of TV shows it distributes to stations across the country. The project was for one section of the corporation, and therefore was not a high priority on the applications-development list. Indeed, Gaites doubts the project would have been completed via the traditional route.

"We had been looking at Burroughs' Linc," recalls Gaites. "The tape-management job, because it was an isolated application, seemed like a good place to try it." A project team began constructing a prototype in January 1983. Gaites and his staff sat with users at terminals and solicited their reactions to suggested routines. "Taking the feedback, the developers sometimes came back with the desired changes within hours," he says. "They could never have made the changes so fast the old way." The system was running three months later.

The prototype for the videotape-management system actually became the "live system," says Gaites. Users were satisfied, and there was no need to put the prototype into production.

Building a prototype for users to play with *before* the application is put into final production heads off many potential problems, says Gary Guttman, president of Generation Science Inc.

A system to administer contracts with radio and television stations was developed by building a prototype and moving it into production. In that case, Gaites' staff built the model system using Data Manager, a microcomputer package from Burroughs, and enlarged its scope in a full-production system interfaced to the corporate-billing application.

Gaites believes the staffer who increased his productivity tenfold is typical. Prototyping has also cut the application-development cycle in half, and that will help his staff reduce Westinghouse's applications backlog. "And if you could put a value on user satisfaction, the payoff would be even higher," he says.

The shortcut as solution

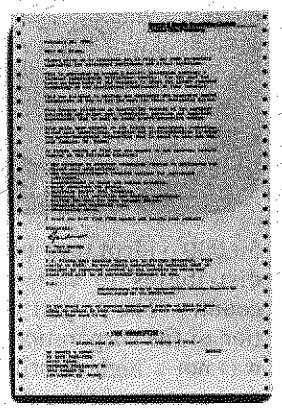
Gaites' experience with the videotape-management prototype becoming a live system apparently was not unusual. AT&T had the same experience with a prototype that system developers were prepared to discard, says R.F. Bellaver. Developers had worked backwards to work up specifications and couldn't even use all the inherent advantages, but the prototype was a hit with the user involved. If developers considered the prototype a failure, the user refused to let it die, claiming it was the "best thing" for his department, recalls Bellaver.

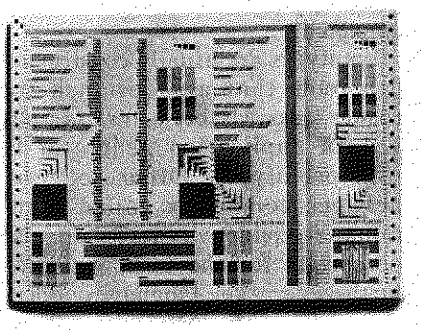
"A user may say, 'Don't bother to do a production version. The prototype does what I want it to do,'" says Joseph E. Urban, associate professor at the University of Southwestern Louisiana. Even if the prototype's benefits are short-lived, users might still be served. Some proponents believe failed prototypes afford users an educational experience. Users involved in their construction learn first hand about the problems that MIS/dp has been trying to explain. For some organizations, the value of this learning experience is worth the cost of wasted time and resources and failed prototypes.

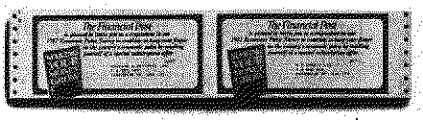
Bellaver believes organizations that use prototyping have a better handle on their work. "There's a bo-

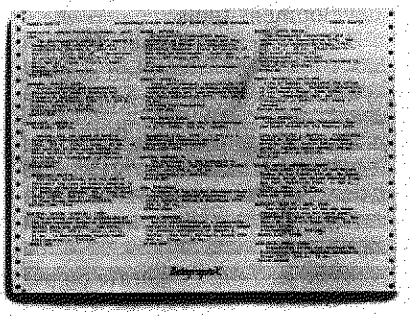
(Continued on page 146)

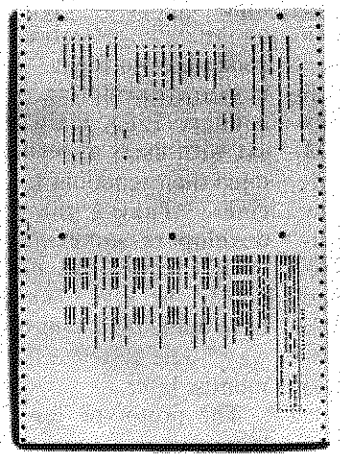
OUR 21,000 LPM PRINTER TAKES MANY FORMS.

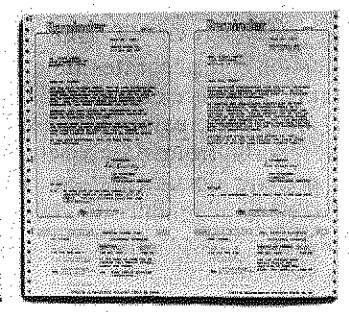


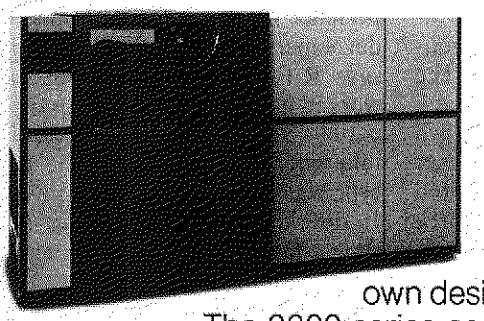


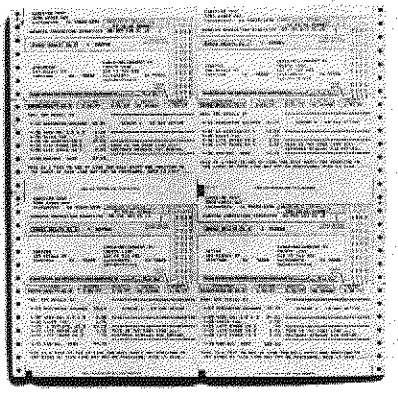


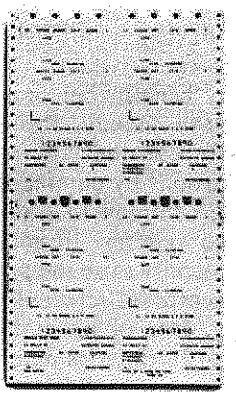












If you need high volume and flexibility, you should know that the DatagraphiX 9800 isn't just a 21,000 LPM laser printer. It also accepts the widest variety of paper form sizes of any non-impact printer, with form widths of 6.5" to 16" and a length range of 3.5" to 14." And all 9800 printers feature perf-to-perf printing on paper weights of 16 to 110 pounds, depending on paper type.

The 9800 series is an entirely new generation of non-impact, high speed laser printers—with more functions, features, and reliability. It offers up to 34 standard character sets, with a font editor that helps you create a nearly unlimited variety of fonts, logos or signatures of your own design.

The 9800 series comes in a variety of on-line, off-line, or on-line/off-line configurations compatible with a broad selection of CPUs. Off-line models offer user-oriented menu-driven software, hard copy log, 6250 BPI tape drives with ping-ponging capability and more. On-line models offer full IBM 3800 compatibility, in addition to the advantages of DatagraphiX's advanced engineering.

Combine these advanced features with excellent print quality and unmatchable reliability, and you begin to see why DatagraphiX is recognized as a supplier of superior computer output management products. The full-featured 9800 printers are available now, setting industry standards for up-time in customer sites throughout the U.S. and Canada.

DatagraphiX®

The Computer Output Management Company
a General Dynamics subsidiary.

Dept. 3515, P.O. Box 82449, San Diego, CA 92138
(800) 854-2045, Ext. 5581
In California, please call (619) 291-9960, Ext. 5581
TWX: 910-335-2058

CIRCLE 62

Prototyping

(Continued from page 142)

nus of ongoing flexibility," he says. His staff builds in what he calls "change hooks," which mark sections of code that might be changed in the future. A year after the prototype is completed, if a user department wants to change elements in the system, Bellaver's staff can locate the pertinent section of code more easily than before. Change hooks improve documentation for updates and revisions. "If we bring in programmers for rewrites, they can grab onto the hooks," Bellaver says. "That's much faster than telling them to search the listings for the right sections of code."

Prototyping also has encouraged Bellaver to approach large-system development in a decentralized, more manageable way. "For years, the MIS/dp community has decried systems so large that nobody fully knew them. With prototyping," Bellaver explains, "we can approach development with smaller teams that understand the projects end to end." The teams work up models in a short time, usually about five months. The result of the team approach is high-quality, efficient projects. "It gives staffers a sense of ownership, improves the quality of their work, and

Selected prototyping tools

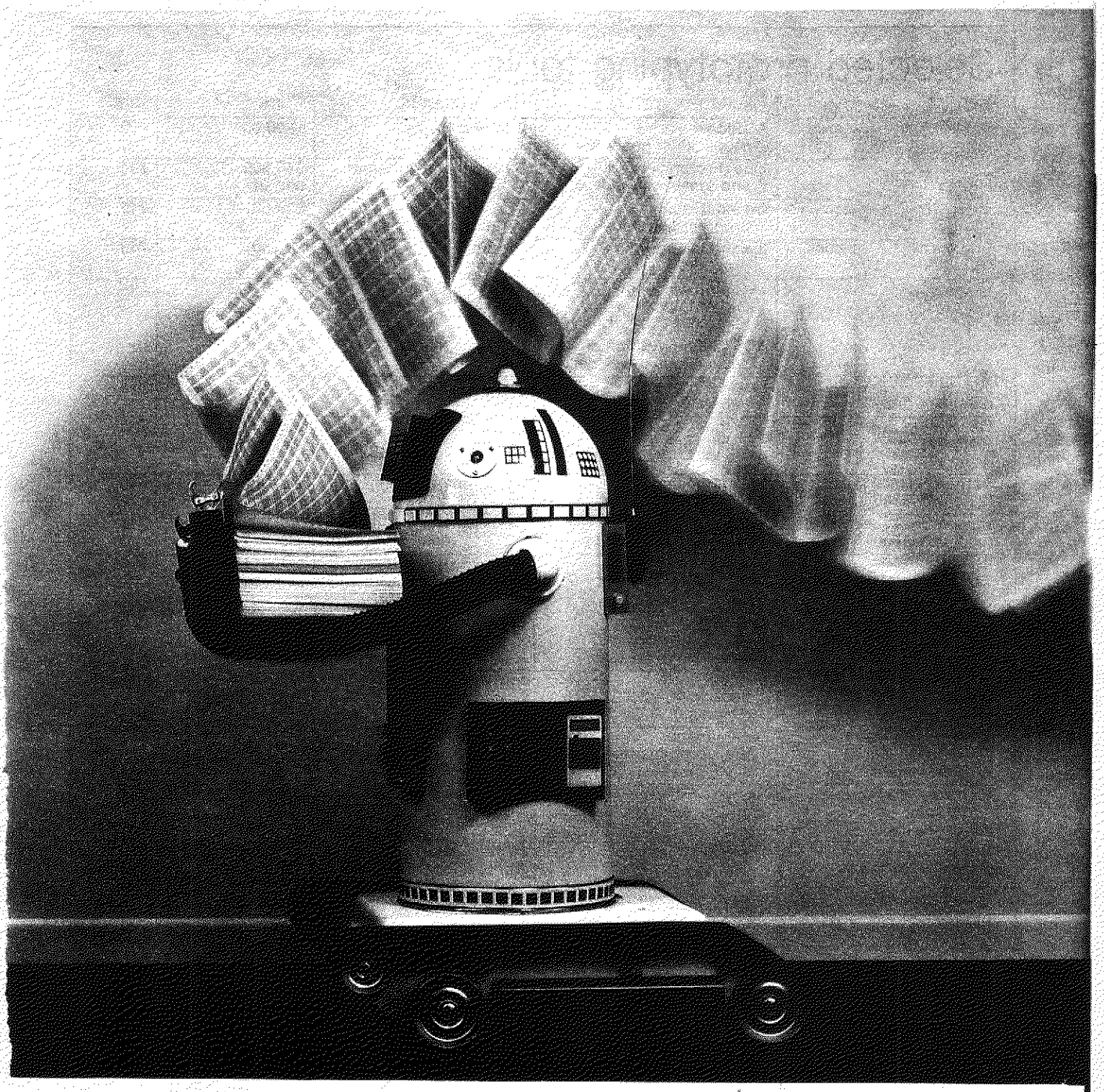
Vendor	Package	Requirements	Price	Circle
Applied Data Research (201) 874-9000	ADRIdeal (Application generator)	IBM and PCMs	$200,000	709
Burroughs (313) 972-7000	Linc (Prototyper and system developer)	Burroughs mainframes	$45,000 and up	711
Bytel (415) 527-1157	Cogen (Cobol program generator	DEC, NCR, Prime systems	$3,650 to $7,500	712
Computing Capabilities (415) 968-7511	Insight (Transaction processor)	HP 3000	$8,500	713
Cullinet Software (617) 329-7700	Escape (Development and database-migration aid)	IBM and PCMs	$25,000	714
	ADS/Online (Database-applications developer)	Same systems	$40,000 and up	
ESI (904) 224-5182	Proxy (Program generator)	Burroughs mainframes	$19,800 to $30,000	715
Generation Sciences (516) 496-3060	Gamma (Cobol application generator)	Many mainframes	$90,000 to $200,000	716
Bernard Giffler Assoc. (215) 343-3345	DPS (Planning and scheduling system)	Sperry mainframes	$25,000 to $40,000	710
Higher Order Software (617) 661-8900	Use.It (System developer)	DEC VAX	$92,000	717
IBM Contact local sales office	Development Management System/CICS	IBM systems	$407 to $685/mo.	718
Index Tech. (617) 491-7380	Excelerator (System-design and documentation tool)	IBM PC XT	$9,500	720
Information Builders (212) 736-4433	Focus (Fourth-generation language)	IBM and PCMs	$66,000 to $130,000	719
Manager Software Products (617) 863-5800	Datamanager (Resource-management system)	IBM and PCMs	$9,000 and up	721
Mathematica Products Group (609) 799-2600	Ramis II (Storage and retrieval system)	IBM mainframes	$45,000 to $90,000	722
Nastec (313) 353-3300	Life Cycle Manager (Project manager)	Nastec Case 2000 workstation	$3,000 $18,000 (including workstation)	723
Software Clearing House (513) 451-6742	Cogen Cobol Program Generator	NCR systems	$3,650 to $4,400	724

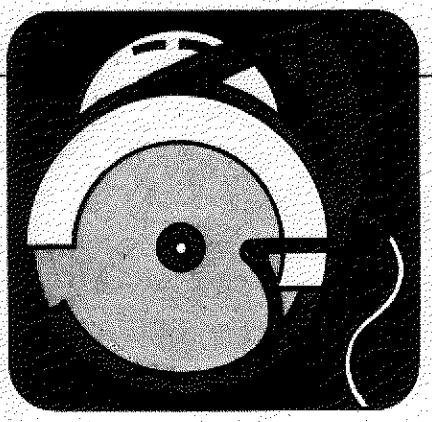
UCCEL

Dallas • London • Paris • Toronto • Frankfurt • Zurich
UCCEL Corporation, Exchange Park, Dallas, TX 75235

Formerly University Computing Company. UCCEL is the trademark of UCCEL Corporation.

CIRCLE 64





inspires enthusiasm," he says.

Not all prototypes are as successful as the ones described by Gaites and Bellaver; many end up in the trash bin. But according to proponents, experimentation is part of the game and shouldn't be considered a drawback. "Prototyping is called quick and dirty, an expression I don't like," says Gary Guttman. "Quick isn't necessarily dirty or bad."

Sometimes most of the prototypes churned out by a development staff are chalked up to experience. Of the five long-distance-telecommunications applications R.F. Bellaver has put through the prototyping route, his staff has discarded three. Does this low batting average distress Bellaver? Not really, he says. His staff's commitments to the discarded prototypes were not heavy. Organizations must be willing to sacrifice a certain amount of time and effort to attain the benefits of prototyping, he says, and those benefits outweigh the false starts.

The inefficiency of building models only to discard them is one of the key concerns about prototyping. Another is how much mainframe computing and storage capacity prototypes will demand. Are prototypes resource-hungry? Yes and no. Says Westinghouse's Ed Gaites: "I haven't found prototypes to be resource hogs. And the systems generated don't require more hardware than usual."

Nonetheless, cautions Joseph Urban, execution time may be slower in a prototype than in a production version. In addition, prototypes sometimes consume more memory capacity than live systems, he says. That's because a prototype isn't a refined product.

Bellaver cautions against becoming preoccupied with start-up costs. "Even if prototyping calls for expensive machines and software, it's worthwhile because faster application development is what management is looking for," he says.

As far as human resources go, prototyping doesn't impose special requirements on development employees. "A normal project team does the job," says Gus Conoscente.

According to Conoscente, Gaites, and Bellaver, managers should avoid several pitfalls when making prototyping part of the development process. Most importantly, don't try to prototype systems that tie together transactions in several departments, they say. Gaites, for instance, chose the videotape-management application to be his staff's first prototype because it involved only one department. Under those circumstances, the development staff had better control over the project and the consequences of mistakes weren't so large.

"Start with something simple and self-contained, something that doesn't have to be integrated into another system," advises Conoscente. "Choose an application of modest size. You and your staff will learn more quickly, and be able to keep your arms around the project. Also, you'll reduce the risk of disaster."

"To get started, choose an application of modest size that doesn't have to be integrated into another system."
Conoscente, Bank of America

If you're looking for a likely place to start, Conoscente suggests a marketing-information system—it doesn't necessarily relate to another automated system. Inventories of real-estate holdings or furniture are other possibilities. Such projects have size and simplicity to recommend them, he says, but they also demonstrate immediate results in functions that are unautomated in many organizations.

Conoscente also urges managers to carefully enlist user members of project teams. The higher the rank of the user members, the better the system will be, he asserts. For example, if you're working on a furniture-inventory prototype, enlist the manager responsible for furniture, not a subordinate. Assign your best analyst to the project as well. Low-echelon staffers don't have a management perspective.

Of course, not all programmers and analysts are suited to be prototypers. They must work closely with users, a role that some technicians find difficult to play. They must have the diplomatic savvy to guide users through the process. In addition, the prototype may imitate a system that's different from the final version, a possibility that users should know about from the start. "Some dp pros are better than others," says Bellaver. "We try to put pros with both programming and interpersonal skills on our teams."

At bottom, prototyping offers managers an instrument to bring the high cost of development—in time, skills, and dollars—under control. But some managers believe prototyping is a vital tool in a distributed-processing setup. Prototyping puts more power and control in the hands of users, helping along the trend toward decentralization. "MIS/dp managers shouldn't feel threatened," says Gus Conoscente. "Their roles are changing. They may become professional consultants within their organizations—or even something better." □

AT&T IS IN ANTIQUES.

Who'd ever think of that?
The people of AT&T Communications.

People who know business.

We can help antique dealers use our AT&T Long Distance Network so that they can hold an auction with different customers in up to 58 different locations.

Anywhere in the United States.

Simultaneously.

So they can find out immediately what the trade is willing to pay for a rare antique bowl.

And any other items in the catalog mailed out beforehand. So their business, in effect, is national.

And they can compete as efficiently as businesses many times their size and sales force.

The people at AT&T Communications are thinking about your business in ways you never thought of.

John Sculley
Chairman of the Board,
and Chief Executive Officer
Apple Computer, Inc.

AppleWorld '87
Los Angeles, California
March 2, 1987

Ten years ago a revolution began that changed the world. It was a revolution that put incredibly powerful tools into the hands and minds of individuals—not computer experts, but ordinary people who were to discover that they could do extraordinary things.

Revolutions require a willingness to take risks...to radically depart from conventional wisdom.

When we reorganized Apple in the summer of 1985, we didn't abandon the legacy of our revolutionary roots. We avoided making the safe choices just to save a company, because the safe choices would have meant losing a dream.

A dream to build great personal computers...a dream to change the world by touching the lives and minds of millions of people, especially the new generations that will inherit the 21st century. We want to be a catalyst for change by improving the way people think, work, organize, communicate, and learn. This is a commitment to action and to changing the ways things are done in the world.

To accomplish this identity for Apple, we have built a work environment where values are shared. Where work is fun. Where creativity and innovation are recognized with ample resources and enthusiastic appreciation.

To create extraordinary tools for ordinary people takes extraordinary talents. We have built a fun, exciting, and rewarding environment, with openness and a shared vision. At Apple we seek to create a great company that—like a magnet—can attract the best people to work for us as employees, or to work with us as independent developers,

resellers, and user constituencies.

It is most fitting, therefore, that AppleWorld™ be a catalyst event that brings together talented people from a wide range of disciplines and experiences across Apple's broad, worldwide constituent groups. The real strengths of Apple are the shared vision we have to make a difference in the world, and the innovative products we build as enabling technology platforms that can be used by others to do wonderfully creative things.

There is no power on earth like an idea whose time has come. As paradigm shifters, we offer the world alternative viewpoints on the critical issues of productivity and learning. As we approach the 21st century, there is increasing evidence that the old ways of doing things aren't working as well as they used to.

In the 1960s, computers were envisioned as wonderful machines that would significantly increase the productivity of large institutions. The idea was to systemize the workflow in the institutions of government, business, and education so that complex tasks could be done much faster.

Yet recently many leaders of institutions are realizing a curious thing: As processing power dramatically increases and at the same time becomes less expensive, we are failing to see a corresponding increase in the productivity of large organizations.

At Apple we see the opportunity to increase productivity through an alternative paradigm. We choose to focus on people rather than on institutions. We seek to turn computers into powerful, easy-to-use tools that will give individuals the chance to discover new ways of learning, working, and communicating with each other. We want personal computers to be a catalyst in the process of discovering new ways for people to do things. By getting people to work better, not just faster, we believe we'll help people to be a lot more productive. And therefore institutions will be more productive.

It comes down to just making the experience of working a lot more interesting. We have barely begun to see how innovative third-party companies and users will be, if we can provide them with an expanding range of incredibly exciting personal tools.

In the process of systemizing work over the last two decades, we may actually have made work boring. It's hard to be more productive if what you are doing isn't interesting. It's hard to be more productive if it's difficult to learn how to use the very tools that are supposed to make work easier.

Just as there have been disappointments about productivity in the work place, there are also doubts about the quality of education in our schools. The traditional view of education has closely mirrored the requirements of an industrial age. We are now past the transition point of becoming a global, dynamic information-intensive economy. The fact is that young people entering this new world can expect to change jobs and even careers several times during their lifetime. Learning must, therefore, become a lifetime experience that is not limited by the boundaries of the institutional walls. Yet education is still conceptualized as a structured, rule-bound process.

I know that there are many progressive educators here at AppleWorld today who believe as I do that the best assurances for gaining a competitive position in the world are directly dependent upon the level of commitment to innovation and resources that we as a society provide to strengthen the quality of our education system.

In an industrial age, most jobs required repetitive manual skills. In an information age, the best jobs will have some information content and decision-making requirement.

If people can work better and more productively, if we make work interesting, isn't there also an opportunity to make the education experience more productive by making learning more interesting?

We believe that personal computers have only begun to be used in ways that can make a difference in education. We see the personal computer as a wonderful, increasingly powerful simulation machine that can be used interactively by the student to learn at his or her own pace, with the capability of customizing to the way that is most effective for the individual. We are inspired by the possibility of new learning concepts built around experimentation rather than memorization. We dream of libraries of knowledge that are at the fingertips of every student, rather than just a collection of books in a building. Libraries of wonderful, high-quality color photographs, high-fidelity sound, and text annotation—all accessible on a personal computer.

The personal computer industry has made it through its slump. We are about to enter a period of exceptional growth and the introduction of many exciting new products. Now more than ever we need a **framework** against which to judge the choices and appreciate the opportunities ahead.

In the 1970s, **data processing** was the central purpose of computers. This was a time of large mainframes locked away in high-security temperature-controlled rooms.

While all of us were conscious that the cost of computer processing power was dropping dramatically, few of us realized the implications it would have on shifting the epicenter of the industry from the mainframe to the network. Today, the giants in the computer industry are scrambling to redefine their product lines in a **distributed processing** model, as opposed to the traditional data processing paradigm.

I predict the epicenter will shift again as we discover that what we really need to do is not just connect networks to computers, but connect information to people. For people to be productive on a connected network, information must arrive in a recognizable and useful form. I believe that by 1990 the epicenter will shift from distributed processing to **document processing**.

We have already begun to see with desktop publishing the power of a typeset-quality document that includes high-quality text and graphics using professional layout tools. But until now, desktop publishing has been a stand-alone product concept.

As people begin to be connected in workgroups, the preparation of documents can be a shared responsibility. But we have set a very high standard of expectation in terms of ease-of-use for stand-alone desktop publishing applications that must be maintained with workgroup solutions. AppleShare™, our new desktop communications product offering, does exactly this.

As the critical mass of stand-alone and connected workgroups grows, so will the market opportunity for new graphics-based software and peripherals for document preparation.

As workgroups have the ability to access information regardless of where it is on the network, documents will take on an increasingly important value based on their timeliness and on the inclusion of real-time information. As artificial intelligence becomes increasingly important, document processing will lead to increasingly **intelligent documents**.

Document processing, however, is not limited to the printed page. It embraces the transference of valuable, customized, and analyzed information into enhanced graphics, text, and layout for clear communications. In fact, some documents may never be printed out, but only read on a display. Conversely, printed documents must have an easy way of being read into computers and indexed as archival records and source material for future documents. I predict that in the future all word processing, spreadsheet, data base,

charting, and communications applications will have a layout and presentation capability. The intelligent document as a metaphor, therefore, is even more important than the document as a physical thing.

Our goal at Apple is to make this progression toward the intelligent document happen in a logical and consistent way. We have been committed to graphics-based systems for years and have carefully built a systems software technology that can grow as communications and high-performance products are added. This is possible only because we have complete control over our computer architectures—which lets us optimize performance without having to make compromises. At the same time, you will see real evidence this week that Apple has actively started to adopt important industry standards, while helping to create others.

What is clear is that the personal computer's role will only become more important as time goes on. With the quantity of information doubling every 2-1/2 years, we will either learn to cope with it or be overwhelmed by it.

High-resolution graphics and superior human interface, two ideas that were considered too radical only a year ago to be accepted as building blocks for the mainstream of personal computing, are now emerging as two of the most important foundation stones for the second-generation personal computer.

This is an example of how exciting technologies, independent of a framework of understanding, are often misunderstood. But the real source of optimism for the personal computer industry ahead is not based solely on new technologies and new products we will introduce here today. We are seeing the convergence of a conscious, genuine need to radically change the ways we work, learn, and communicate, so our institutions can be more productive and our people better educated, and so a strong, affluent, middle-class marketplace will survive and thrive in a far more complex, global, dynamic economy.

As graphics-based technology moves into the mainstream because it is now clearly needed by the users, we expect to see a tremendous surge of activity by third-party companies that will be creating software applications, peripheral products, accessory products, and communications products for our second-generation personal computers.

So we see a framework with the epicenter of the computer industry progressively shifting from **data processing** to **distributed processing** to **document processing**. A

framework where documents will become increasingly important as more and more people are connected together over networks. A framework where documents themselves will become increasingly more intelligent as data can be easily accessed from anywhere, and as we move toward increasingly higher levels of performance with personal computers that can eventually handle expert systems and artificial intelligence applications. A framework where the second-generation personal computer will play a very significant role in merging high-performance functionality with a superior graphics-based human interface.

It is within this framework that the plans and opportunities for Apple over the coming years can best be understood.

Today you will hear about a large number of very significant products from Apple and third-party developers that are being announced at AppleWorld '87.

For the office market, what has been the high end of our Macintosh™ line suddenly becomes the low end of that product line.

We have expanded the performance of Macintosh while now offering a choice of compatibility with other operating environments as an option.

We are announcing a significant development for AT&T® UNIX® that will give high-end users the opportunity to have the same outstanding human interface on UNIX® that many have become accustomed to using with Macintosh.

AppleShare and the related workgroup productivity communications products that we announced in January will now have an even wider range of choice of file servers, since AppleShare products are fully compatible with the new Macintosh computers that we will be announcing today.

The expanding range of Apple's product line, including the ability to use industry standards such as Ethernet and Token Ring to connect into host environments, will open up new opportunities which we intend to pursue with the federal government and high-end resellers.

But in spite of all you will see and hear about this week, it is important to keep in mind that our pipeline is still filled with even more new products to come. Before the end of 1987 there will be more announcements, and still more in 1988.

We are even making significant progress in the development of the technologies that we hope to be able to use in our products in the early 1990s.

We are a new products company in a new products industry, and never again will we allow Apple to get behind the product-development power curve. Staying ahead of the product-development power curve is very important for Apple in terms of the positioning of our company, in terms of assuring strong third-party innovation with our technologies, and in allowing us to maintain gross margins high enough to support the significant research and development expenditures necessary to support two system software architectures...Apple® II and Macintosh.

Apple is a company of meaningful differences, not better sameness. We intend to be the catalyst in this industry—the catalyst that will provide the best opportunities for others to innovate with our products, and will offer still others the chance to pioneer new markets and new methods of selling. We strongly believe the future health for the entire personal computer industry lies in being able to add clear value, not in turning the industry into a cookie cutter operation for clones.

Last year people asked, "When will Apple be able to connect to them?" Next year, I predict the question will be, "When will they be able to connect to Apple?"

The second-generation personal computer is here. But the second-generation personal computer is not a box. It's a consistent set of building blocks that form a systems software architecture—an architecture that retains the elegance of the stand-alone Macintosh and Apple IIGS™ in "look and feel," yet has the industrial strength to be able to work in a serious network and data communications world.

Not long ago, conventional wisdom held that Apple had embarked on a lonely and dangerous course by investing so much in the Macintosh technology.

Today, as the rest of the computer industry bumps up against the ceiling of ordinary computer technology, Apple is just beginning to realize the potential of our original idea.

While our competition's whizziest 32-bit machines can only run the same software slightly faster, Apple computers are setting new standards of speed and performance with totally new kinds of software.

Our revolutionary idea isn't a silicon chip or a clever twist on technology. It's a vision of how computers can help people accomplish anything they set their minds to do. Simply put, it's **"the power to be your best."™**

And when you think how much that vision has changed our world in the past decade, just imagine how far it can take us in the next.

Apple and the Apple logo are registered trademarks of Apple Computer, Inc. AppleShare, Apple IIGS, AppleWorld, Macintosh, and The Power to Be Your Best are trademarks of Apple Computer, Inc.
UNIX is a registered trademark of AT&T.

The object-oriented paradigm is a programming method gaining much commercial attention from those who must build advanced computer systems. It promises tremendous advances in programmer productivity compared with traditional structured programming techniques, because its basic unit is the self-contained object, which combines data and algorithm. Proponents claim that object-oriented programming is more "intuitive" than previous methods and therefore is easier to work with. With processing power and memory priced so low, a number of companies, large and small, are creating a market for object-based tools.

The OOPS Revolution

BY JOHN W. VERITY

Just down the road from the general store in Sandy Hook, Conn., past a sweeping turn in the Pootatuck River where all that could be heard was the rush of a waterfall and the rustle of pine trees, there sat, many years ago, a red brick building that saw the beginnings of a technological revolution.

It was there in the mid-1800s, it is said, that Charles Goodyear worked to perfect his accidentally discovered process for vulcanizing rubber. Vulcanization turned raw sap from the Amazon into a tough, versatile product that eventually was used in the manufacture of automobile tires. Goodyear's discovery thus transformed an entire industry.

APPLE'S LISA AND MACINTOSH ARE OBJECT BASED.

The general store, circa 1831, is still there, selling everything from hammers and nails to chocolate chip cookies. The old brick building is there, too, and one can still hear the pines whispering and the waters spilling over rocks outside. And, if Tom Love, chairman and cofounder of Productivity Products International (PPI), is to be believed, there's another revolution fomenting behind those brick walls, one that is about to change the way another industry does business.

This revolution has no Lenin as its fiery leader—Love's flashy white BMW betrays strictly capitalist tendencies—nor does it need one. This is a movement of the people, and a worldwide one at that. From Oslo to Tokyo to Palo Alto, the growing masses, their consciousness raised at international conferences, are saying things that would make Marx spin in his grave if he could hear them. "Class structure," they demand. "Inheritances for all."

If Love and others at the vanguard of this revolution get their way, class and inheritance will in fact become vital issues for software builders everywhere. Those are concepts key to the practice of object-oriented programming, a technique fast becoming commercialized by Love's venture-financed PPI, and by such firms as AT&T, Xerox, Tektronix, and Apple. These vendors and a growing list of large users claim that object-oriented programming systems (no one calls them OOPS yet) are already making good on the promises that the last great leap forward—structured programming—continues to make but has yet to fulfill.

Although structured methods have clearly extended the possibilities of computer programming, they have been less than satisfactory when dealing with large, complex systems, particularly those that are highly interactive and whose specifications are therefore difficult to pin down early in the development cycle. Backlogs continue to grow, bugs proliferate, deadlines are missed, maintenance is as tough as ever.

"Structured programming was only a small help," states Love. "It provided only a 10% to 15% improvement in productivity when people were really looking for improvements of 10 to 15 *times*." Object-oriented programming tools, he claims, stand ready to provide those needed improvements because they offer means for a radically different, more "intuitive" way of conceptualizing and building systems.

Instead of decomposing systems into hierarchies of nested boxes as orthodox structured design methods do, the object-oriented way is to simulate the

world in terms of cooperating "objects" that relate by passing strictly defined messages to one another. "It's program as stage play," states one convinced user.

"Object-oriented programming is the structured programming of the '80s," says Bjarne Stroustrup, a respected software thinker at AT&T and author of its object-oriented language, C++. "In the right hands it is a major lever."

"It's significantly more of a revolution than structured design, because it applies to a wider variety of applications," comments David Thomas, associate professor of computer science at Carleton University, Ottawa.

Vision of a Global Market

Of course, the object-oriented gang may sound less than, well, less than objective when they claim their approach improves programmers' productivity by factors of 25 or more, or that it is more easily used by "lesser trained programmers," or that it finally makes possible the reusable software "components" the industry has coveted for so long. Perhaps most fantastic is the vision some share of a potentially global marketplace for software components analogous to today's microchip arena. Love's PPI has already trademarked the term "software-IC" and is seeking partners to help build such components.

Surely the multibillion-dollar software business has seen more than its share of snake oil and wonder cures before. Yet, the fact is that a growing number of companies, some of them backed by substantial capital and resources, are successfully hawking object-oriented tools. There is serious talk among software theorists, moreover, that object-oriented techniques will fundamentally change the way computer systems are conceived and designed. Large companies whose products depend heavily on complex software—companies like LM Ericsson, AT&T, ITT, and Hewlett-Packard, to name but a few—have adopted the technology in one form or another. Revolution or not, something is going on.

Tangible evidence of that something was seen in Portland, Ore., last October when the first conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) was held. Some 20 vendors showed up for the ACM-SIGPLAN sponsored conference, more than 50 papers (only a third of those submitted) were delivered, and some people hoping to attend had to be turned away for lack of space (just over 1,000 people got in). The meeting's organizers, who represented everyone from IBM to the Central Intelligence Agency, were surprised by all of this—but only a little.

"A lot of people have been working with object-oriented programming for a while now and this conference was overdue," says OOPSLA program chairman Daniel Ingalls, a principal engineer at Apple Computer Inc. "The fact that we had to turn people away shows that the area is really growing."

"It's an unstoppable trend," declares even Tom DeMarco, a noted advocate of structured methods who is founder and a principal of the Atlantic Systems Guild in New York.

While similar conferences have been held recently in France and Britain, there are other signs of object-oriented programming's increasing acceptance. Digital Equipment Corp., for example, has set up an Object-Based Systems Group in Hudson, Mass., where it does much of its work in artificial intelligence and advanced programming.

SMALLTALK BINDS DATA AND PROCEDURAL CODE TOGETHER.

Another big industry name, Xerox, has spun out a new company, ParcPlace Systems in Palo Alto, which will create and market a variety of software tools centered around Smalltalk. That language, developed over the past 15 years at Xerox, is considered by most observers to represent the archetype of object-oriented programming. Xerox has licensed Smalltalk to such firms as Tektronix, Apple, HP, and Sun Microsystems for use on various workstations and is seeking more such partners, according to Adele Goldberg, ParcPlace president.

There's been activity on campus, as well. At Brown University, IBM has helped fund a multimedia database project whose programming is object oriented. The so-called Intermedia system combines texts and illustrations into a "web of information" that can be browsed, annotated, and shared among teachers and students using workstations. It is interesting to note that Brown programmed IBM RT in part with software migrated from Apple's Macintosh.

Dave Patterson, a professor at the University of California, Berkeley, has designed a reduced instruction set computer (RISC) microprocessor designed specifically to run Smalltalk. It's called SOAR. Farther up the coast at Oregon State University in Corvallis, computer science instructor Timothy Budd reports that the first printing of his book, *A Little Smalltalk* (Addison-Wesley, Reading, Mass., 1987), has sold out. He has also sold about 300 copies of his Unix-based teaching version of the language, which goes by the same name.

In nearby Beaverton, where Tektronix sells a line of Smalltalk-based workstations, Servio Logic Development Corp. markets Gemstone, an object-oriented DBMS for personal computers and VAXs. Computer Corp. of America, Cambridge, Mass., is understood to be working on similar products.

Digitalk Inc., Los Angeles, sells for $99 an impressive, homegrown version of Smalltalk for the IBM PC family. The software has found extensive use at Olivetti, the Italian computer maker.

Back in the States, Tom Love's PPI claims to have installed some 2,500 copies of its Objective-C package, a preprocessor that feeds standard C compilers with object-based code. It also sells Vici, which interpets C and Objective-C code for debugging and instructional purposes. Customers include HP, Accuray, and NASA.

A Product with Pluses

AT&T is gearing up to push C++ into the commercial market after seeing it find use at some 200 universities worldwide, according to Zach Shorer, product manager in Morristown, N.J. Shorer claims that over a million lines of C++ code exist within AT&T alone, and that the language, even without much push from the company, has been adapted to machines ranging from Unix workstations to Amdahl mainframes; a Cray supercomputer version is in the works. Bjarne Stroustrup, the language's modest author, says his C++ is "spreading like wildfire" within AT&T, where it is used for, among other things, simulating VLSI chips.

Key Logic, a Santa Clara offshoot of Tymshare, will soon introduce an entirely new operating system for 370-type mainframes that completely replaces MVS and competes in terms of transac-

tion processing speed with IBM's TPF/2 (the former ACP). The software has been written in an object-oriented version of PL/1, according to Key Logic president Ann Hardy. "We couldn't have built the system without it," she says.

The CIA has commissioned Xerox to build a system called the Analyst, a "multimedia spreadsheet" that enables intelligence analysts to peer into numerous textual and graphical databases at the same time. The hush-hush system is composed of objects.

"Best of the Known Techniques"

Enthusiasm abounds. "We've been convinced since 1975 that this is the best of the known techniques," says Carleton University's David Thomas, who was given a private showing of Smalltalk back then. But as he and others are quick to point out, it's only been recently that object-oriented programming has become commercially feasible, because it requires substantial computing resources to be effective. Prices for memory and processing power have reached low enough levels, particularly in the form of desktop workstations, that now the basics of the object-oriented technique can be taught and used throughout industry and academia. "Now there are a lot of people beavering away at this technology," he says.

"It's definitely the wave of the future," states Paul Cubbage, senior analyst at industry researcher Dataquest in San Jose. He doesn't think, however, "that it will be at the center of the market until the 1990s."

Indeed, even Love, who's raised close to $3 million in venture capital to launch PPI, believes the new methods and languages have a long way to go before they dent COBOL usage to any extent. "We won't go bust our pick on that," he replies when asked if PPI intends to try for the mainframe market. Instead, the company plans to attack the applications backlog at the "fringe," where entirely new, advanced, and often complex systems get built. Insurance companies that have to build their own multimedia databases, for instance, will be forced to adopt object-oriented tools if they want to succeed, Love maintains.

If there really is an object-oriented revolution going on, Smalltalk is its manifesto. For most people, it is the first thing that comes to mind when the subject of object-oriented programming is broached. A product of Xerox's Palo Alto Research Center (PARC), Smalltalk was the brainchild of the company's brilliant computer scientist Alan Kay. Kay's precocious vision in the early 1970s was of a notebook-sized computer with which children and other nontechnical users could interact graphically through a display of two-dimensional objects. The objects would reflect the machine's internal state and could be manipulated to change that state.

Dynabook, as Kay referred to his laptop dream machine, could not possibly be built from the hardware or software available back then. Nevertheless, he and his team of researchers (which included Ingalls and Goldberg), built and simulated as much of it as they could. The fruits of their labors were a dazzling series of innovations at PARC that helped yield, among other things, Xerox's Alto and Star workstations, the mouse-icon-window-bit-mapped-screen display as the leading idea in user interfaces (popularized by the Macintosh), and Smalltalk as a way of life.

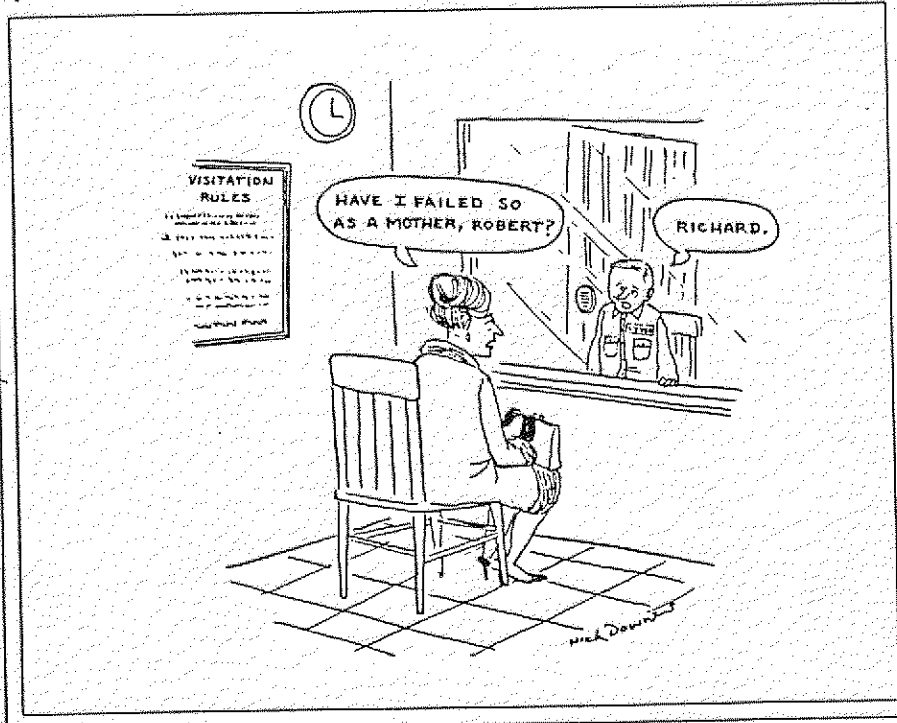
Well, almost. Smalltalk was not just another programming language, one quickly learned; it was a complete programming "environment" unlike anything anyone had ever seen before. It provided an abstract world in which the usually distinct boundaries between program and operating system and between data and program were blurred, a world in which a new conception of programming could flourish.

Primary Cause of Project Failure

The very name of the data processing industry reflects the paradigm of procedural programming, as seen in FORTRAN and COBOL programs, for instance. Data are structured in some way in order that they may be processed by a separate and shifting collection of procedures. But since there is no firm connection between these two elements (they are stored "without context," Love explains), a change in data structure can easily invalidate some or all of a program's routines, and vice versa. This, say software theorists, is the primary cause of failure in large, complex projects where no single person can comprehend both data structure and program logic in their totality. Traditional languages force programmers to rely on potentially faulty assumptions about which data types are valid for which routines.

As a result, fixes, patches, and extensions to a program tend to produce unwanted, unpredictable side effects and even catastrophic failures which, so far, no amount or form of structured methodology has been able fully to prevent. "At about 100,000 lines [of code], things start to break down with the old methods," states Love. "As your ability to understand a system declines, your ability to add to it declines as well."

In contrast, Smalltalk and related languages bind data and procedural code tightly together—inseparably, in fact—in the form of objects. Each object contains its own data, appropriately struc-



The OOPS Revolution

tured for its particular use, to which it has sole access and sole responsiblity for manipulating according to private procedures. Thus, there's no chance for data and code to get out of sync with one another, as it were, and there is greater possibility for building flawless programs quickly and in a way that permits almost endless extension and change. With relative ease, objects can be put together to form new systems and extend existing ones. "You don't destroy the original [code], you just extend it and build upon it," explains Jim Anderson, president of Digitalk.

Objects are quite different from traditional subroutines. For one thing, they reflect a deeper abstraction: by strictly tying data to code, objects actually maintain structure and context even as they helpfully "hide" those qualities from the programmer. Moreover, they respond only to certain strictly defined messages passed to them by other objects. Once it receives a message it knows how to handle, an object takes full control of the system until it passes control to some other object via another message. (In contrast, traditional "well-structured" subroutines eventually pass control back to a main routine.) The programmer need not know anything about an object's internal structure, either its data or procedures, yet he can still use the object if he knows the messages it can receive and act upon.

Finally, the object's interface to other objects is clearly defined and cannot be subverted: a Smalltalk program cannot jump suddenly into the middle of an object in the way that a wayward FORTRAN program might mistakenly activate code deep within a subroutine. This strictly defined interface, combined with the fact that data and procedures are sealed away from dangerous tinkering, prompts talk of a future marketplace of software objects, designed to be cataloged and available off the shelf from networked libraries of components.

Already, Artecon Inc. of Carlsbad, Calif., sells a family of basic graphics objects designed to work in PPI's Objective-C environment. Love expects additional sets of useful objects to become available in such areas as communications, databases, and user interfaces.

In addition to their encapsulated nature, objects gain much power from their ability to *inherit* properties and behavior. All objects reside in a many-leveled hierarchy of *classes*. Each object is considered an *instance* of its class and, like other instances thereof, it displays the properties and behavior of not only that class but also those of all classes above it in the class hierarchy. Thus, the object "Secretariat" might be an instance of the class "horse" and inherit properties and behavior from the class "mammal," which itself would inherit from the class "animal." Inheritance makes it possible to define complex new objects without the bother of writing everything from scratch.

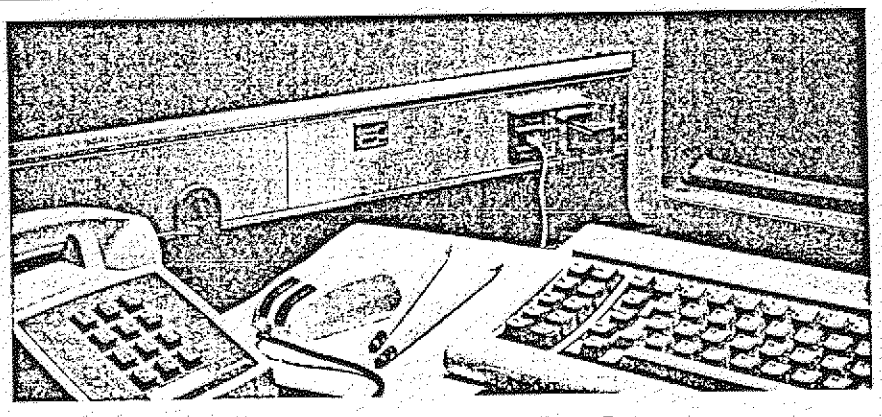
Means to Tremendous Improvements

Although objects and the notions of inheritance and class take some getting used to—several months, at least, for traditionally trained programmers, it is said—they provide the means for achieving tremendous improvements in productivity. For one thing, large collections of predefined, well-documented classes and *methods*, the algorithms in an object that act on incoming messages, can be delivered for programmers to use as is. Such predefined classes may be quite sophisticated, defining frequently used objects like interactive windows or disk browsers. In addition, user-defined classes and objects may be stored away for future use and adaptation.

Interactive systems tend to employ sophisticated, graphical interfaces that display a high degree of modularity and are therefore more easily programmed in terms of objects, says Beau Sheil, a Xerox PARC alumnus who now heads Price Waterhouse's newly formed Technical Center in Menlo Park, Calif. (see his "Power Tools for Programmers," February 1983, p. 131).

ParcPlace's Goldberg notes, too, that the notion of objects can be easily grasped by nonprogrammers, who can "define their own applications and prototypes" in terms that are familiar to them.

"The world we see around us is made up of objects, not subroutines," says one proponent. But does that make for a revolution, a replacement of structured design methods? AT&T's Shorer thinks so. "We have a tiger by the tail," he says. But Oregon State's Budd takes a more cautious view: "Object-oriented programming is just part of a bigger revolution. People are realizing that we have a plethora of languages available because all language paradigms are important." ■



Manage power, telephone and data wiring in your facility. Free brochure shows how.

More telephones, a new computer, new copy machine, new electrical equipment—wires, wires everywhere. A Wiremold aluminum raceway system efficiently tucks away those wires and cables into one low-profile, surface-mounted raceway. It can run around the walls or along a counter top. The satin anodized aluminum finish looks great, and it's easy to keep that way. Talk to your installer about Wiremold. Or write/phone for the brochure. 1-800-621-0049. (In Connecticut) 1-800-992-2277.

Wiremold®

The Wiremold Company, Electrical Division, 60 Woodlawn St., West Hartford, CT 06110-0639.

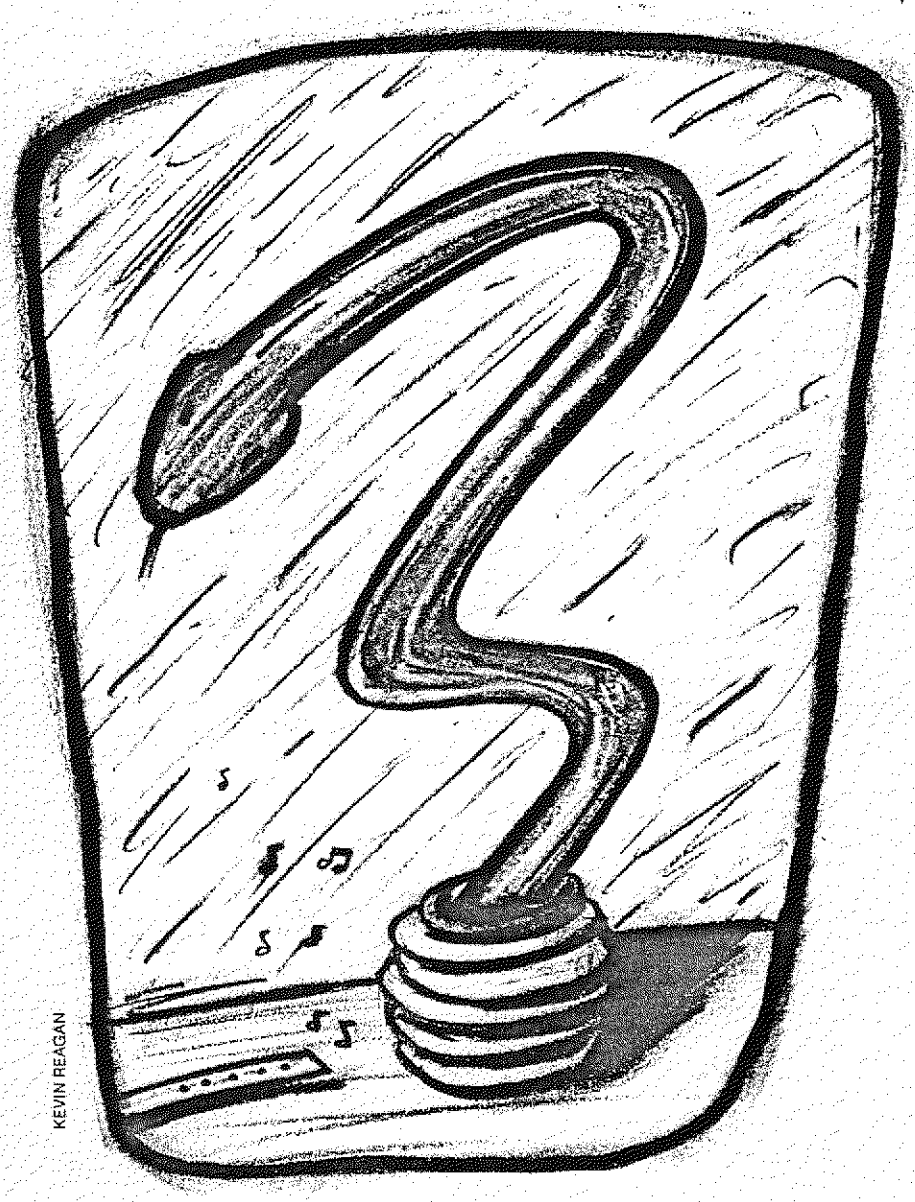
CIRCLE 39 ON READER CARD

AC: FYI R.R.

Experimental Prototyping in Smalltalk

Jim Diederich and Jack Milton
Mathematics Dept., University of California at Davis

Smalltalk promotes fearless programming. Users can significantly alter an application — even the system itself — without fearing unrecoverable disasters.



KEVIN REAGAN

Object-oriented programming is one of the central programming paradigms to emerge in this decade. The scope of its influence can be seen in the introduction of objects into numerous programming languages such as Loops, Objective-C, Object-Pascal, and Flavors, among others. Object-oriented concepts are also being incorporated into existing relational database technology. Database management systems now exist that are based on object-oriented principles, and one commercial version uses an extension of Smalltalk for its data definition and data manipulation languages. Additional uses and applications can be found in the proceedings of recent conferences.

Smalltalk is the principal object-oriented language. While a more complete discussion of Smalltalk's development is given in the so-called Green Book[1] and in the August 1981 issue of *Byte* magazine, it is important to note that Smalltalk is a descendent of Simula and has its origins at the Xerox Palo Alto Research Center in the early 1970s. It was developed by the Learning Research Group based largely on ideas of Alan Kay.

The language went through three major versions: Smalltalk-72, Smalltalk-76, and Smalltalk-80, which was first licensed for use in 1983. Commercial versions of Smalltalk are available on such workstations as the Tektronix 440X series, the Sun, and the IBM PC AT. Prerelease versions have been made available for the Apple Macintosh. An initial attempt at developing Smalltalk on minicomputers such as the Digital Equipment Corp. VAX line was judged unsuccessful in a multiuser environment. However, implementations for the MicroVAX are under development.

Smalltalk is more than just another programming language. It offers a completely new environment for software development. Many articles and books have been

0740-7459/87/0500/0050/$01.00 ©1987 IEEE

written describing the features of the Smalltalk language and environment, but our primary objective is to convey just how *different* software development is in the Smalltalk system.

Our experience, based on our work in prototyping a database design system[2] over the past year suggests that Smalltalk goes well beyond facilitating programming. Indeed, it is an integral tool for promoting experimental prototyping. More explicitly, in our experiments we often made sweeping changes in the design system's architecture, generally with little reprogramming effort, and usually with the introduction of only simple bugs that were easily identified and fixed. To a large extent, our desire to try different approaches was significantly influenced by what we view as a new and emerging concept in programming.

Basic concepts

Two important aspects of the object definition in Smalltalk are encapsulation and hierarchy.

In encapsulation (or information-hiding), an object has its own data or local memory, called instance variables. An object also recognizes a set of procedures for manipulating its local memory. These procedures, called methods, are invoked by sending messages to the object.

Naturally, many objects will have the same type and will respond to the same messages. Consequently, objects are organized by a hierarchy of classes and subclasses. An object responds not only to messages defined for its class, but it can also inherit messages from all its superclasses.

Class definition. Figure 1 shows the definition of two classes, Person and Student. Person is a subclass of Object, which is the root class of all other classes. An instance of the class Person (an object from the class) will have local memory for its name, address, and birthDate. Student is a subclass of Person, and each instance of Student will inherit name, address, and birthDate, in addition to having instance variables college, class, major, currentCourses, and gpa.

Each instance variable will be an object, too, but it need not be bound to a particular object class. HonorsList is a class variable. Every method of the class Student will have access to (can read or change) the value of this variable.

(Note the naming conventions: Compound names have the first letter of each word capitalized except for the first word.

Encapsulation and hierarchy are two important aspects of object definition.

The case of the first letter of the first word depends on the use of the name. Thus an instance variable for a graduate student would be graduateStudent while a class or a class variable would be GraduateStudent.)

Messages. After creating the class Person, we can develop messages that can be sent to the class Person and to instances of the class. For example, the existing message, new, is inherited by the class Person from its superclass Object and will create an instance of Person. In Figure 2a, the temporary variable person is assigned (using the assignment symbol ←) and serves as a pointer to the object that is created as an instance of the class Person.

To give an instance a value for its name, we can create a message, name: aString, that can be sent to person, as Figure 2b shows. The object being sent the message, person in this case, is called the receiver of the message. Messages that have colon suffixes, such as name:, take an object as an argument, and those without colons do not.

A message can contain several parts, such as add:before:, which in this form, without arguments, is called a message selector. It can be used to add a new object before the current object in a list, as Figure 2c shows. Likewise, since Student is a subclass of Person, instances of Student can also be sent the message, name: 'Stefano'.

A subclass can also reimplement and thus override a message defined in its superclasses. For example, the message, passing, sent to a student might determine if the student's GPA is above 2.0. However, if we create a subclass GraduateStudent of Student, we can add a message also called passing for this subclass that instead determines if the GPA is above 3.0. Thus, if an instance of GraduateStudent is sent the message passing, the system looks to this class and finds the message with the 3.0 condition. Likewise, if an instance of Student is sent the message, the system uses the message with the 2.0 condition.

If we changed our minds and later wanted to use the same condition for both, we would delete the message for Graduate-

```
Object    subclass: #Person
          instance variables: ' name address birthDate '

Person    subclass: #Student
          instance variables: 'college class major currentCourses gpa '
                class variables: 'HonorsList '
```

Figure 1. Class definitions.

```
(a)  person ← Person new.

(b)  person name: 'Stefano'.

(c)  personList add: newName
     before: currentName.

(d)  name: aName
     name←aName.

(e)  name
     ↑name
```

Figure 2. (a) Creating an instance; (b) message modifying an instance variable; (c) message with two arguments; (d) method for the message name:; (e) method for the message name.

Student. Then whenever an instance of GraduateStudent is sent the message passing, the system first checks its class, and when the message is not found there, it goes up through the hierarchy to find and execute a message with that selector.

Methods. Messages are implemented in routines that are called methods. For example, the method for assigning the name of an instance of Person is quite simple, as Figure 2d shows. The first line of a method always gives the form of the message. In Figure 2d, name: is the message selector and aName is an argument. The remaining lines implement the method. In the second line the instance variable, name, of the message's receiver is assigned the value aName.

Similarly, Figure 2e shows the method for the message used to retrieve the person's name. Again, the first line of the method is the form of the message. In the second line, the instance variable, name, will be returned, as signified by the up arrow (↑), whenever the message, name, is sent to an instance of Person.

Every time a message is sent to an object, something is returned. The returned value may be significant or may merely inform the sender that a requested action is complete. This lets messages be concatenated. For example, person birthDate month, will return the month of person's birth since person birthDate returns a birthDate that has an instance variable month that is returned when the message month is sent.

Variables and scoping. To enforce the concept of encapsulation, there is only one way to modify an instance variable of an object — sending the object a message. Furthermore, if a variable appears in a method, it can only be one of six types:

- an instance variable in the class of objects for which the method is defined,
- an argument of the message,
- a temporary variable local to the method,
- a class variable,
- a pool variable, or
- a global variable.

Class variables are shared by a class and its subclasses, pool variables are valid across designated classes, and global variables are shared by all classes.

In the method for name: in Figure 2d, name must be an instance variable because it is lowercase and it is in a method defined for instances of Person that have that instance variable. The variable, aName, is an argument of the message. There are no temporary or other types of variables in this method

To enforce the concept of encapsulation, there is only one way to modify an instance variable of an object.

Class, pool, and global variables are used sparingly. Thus, in practice, we have a very restricted type of lexical scoping. This strict information-hiding all but eliminates scoping-related problems, and names typically do not have to be modified to avoid naming conflicts.

For example, consider the method (in Figure 3) defined for instances of Person that when sent to a person, the receiver, switches its name with that of aPerson, the message's argument. Temporary variables, such as tempName in Figure 3, are declared by listing them in the vertical bars. They exist for the message's execution life. (Class and global variables are used for longer term storage and are not in the local memory of instances of the class.)

Note the various uses of name. In the first line of code, it is a message sent to aPerson to retrieve aPerson's name, and assigned to tempName. Because aPerson is not the receiver of the message switchNamesWith:, the only way to retrieve its name is to send it the message, name.

The next use of name is as the argument of the message name: in the second line of code. In this case, because it is not declared a temporary variable, it must represent the instance variable of the message's receiver. In this line, the name of aPerson is modified to be the name of the receiver.

In the last line of code, name is the instance variable of the receiver and is changed to the name in tempName. The code in Figure 3 could have been written with different names for the message selectors, such as getName instead of name for retrieving a name and setName: instead of name: for storing a name.

We chose this example to illustrate the freedom permitted by the Smalltalk language; and this overloading of name presents no difficulty in understanding to the slightly experienced Smalltalk programmer.

There are some negative aspects to using getName and setName:. First,

```
aPerson getName
```

is procedural in flavor while

```
aPerson name
```

is more functional and has more of a natural-language flavor, which contributes to the readability of the code.

Second, it precludes the simple convention that messages with the same names as instance variables are used to retrieve their values, while those followed by a colon and argument are used to store the values. This convention makes remembering the message selectors for instance variables straightforward.

Control structures. Smalltalk control structures are also handled via the object-message paradigm. For example, a message, do: aBlock, can be sent to collections of various types to process each element. For example, the code in Figure 4a will process each student in classList. The argument of the block has a colon prefix, appears before the vertical bar, and is in turn instantiated to each of classList's elements.

```
switchNamesWith: aPerson

|tempName|

tempName ← aPerson name.
aPerson name: name.
name ← tempName.
```

Figure 3. A method in class Person.

The code following the bar is executed for each element in the list and illustrates the use of a conditional. The condition, an instance of the class Boolean, is enclosed in parentheses and is sent the message ifTrue:ifFalse:. Other conditional message selectors are ifTrue:, ifFalse:, and ifFalse:ifTrue:.

Understanding objects. When working with objects, an individual accustomed to non-object-oriented languages may experience some unanticipated difficulties. Most can easily be corrected once the symptoms are recognized.

For example, there is a tendency to take the object paradigm too literally and consider each appearance of an object as a unique object. Certainly in the real world, an object can only be in one place at a time. Suppose an instance of the class Card has the instance variables suit and rank. A hand is an Array that can hold as many as five cards, and a deck is an OrderedCollection of 52 cards.

The code in Figure 4b will deal a card and place it at the first position in the hand. If you now inspect deck and hand using the inspector in Smalltalk (a tool for examining objects), you see that deck still contains the dealt card — and so does hand. A literal interpretation of object, that an object can only be in one place, would suggest to some that these are distinct cards. Unfortunately the inspector does not directly reveal whether they are distinct or not.

Assuming they are distinct might lead to an attempt to set the suit and rank of the card in the deck to nil to avoid redealing it, which will result in making the card in hand a blank card also. What in fact is occurring is that the assignment statement places a pointer in the variable aCard to the same card in both collections, deck and hand. Indeed, the card exists in one place, and aCard, deck, and hand merely provide alternative access structures that point to it.

In some cases, the problem is not so easily diagnosed, particularly when there are several layers of complexity such as when using the model-view-controller triad. Khoshafian and Copeland discuss different degrees of object identity.[3]

Fearless programming

While it is difficult to capture in an article the quite different sense of what it is like to work in an object-oriented language and environment, we will nevertheless try to present some sense of the flavor of prototyping in Smalltalk. What we would like to show, but can do so only partially, is that there is an undercurrent we characterize as fearless programming.

Fearless programming encourages experimentation with alternative approaches to algorithms, application programs, and system design without the fear of being caught up in a morass of detail that is too painful to sort out. Moreover, the system has a robust programming paradigm — the programmer can make bold changes to the system itself and, even after making significant errors, can often proceed through several different routes to recovery without harm.

In fact, fearless programming has played a large part in our development of new methods and faster algorithms for database normalization because it promotes experimentation. (We do not mean to imply that good design can be ignored or that good programming practices can be violated, for even Smalltalk methods can be created that have unanticipated side effects or fail to be coherent and properly modularized.)

Advantages. Naturally, the closer the constructs in a language are to the entities we deal with in the real world the less difficulty we encounter in translating the real-world problem into a program. Object orientation is a major step in this direction, since working with objects seems more natural than working with constructs found in standard languages.

For example, in a non-object-oriented language, if an element is retrieved from a list or an array, as in the assignment statement $x :=$ array(5), then changes made to x are not reflected in the contents of array(5) and vice versa. This assignment in effect creates a duplicate element, and duplicates can easily lead to inconsistencies, since changes in one element are not automatically reflected in changes in the other.

This is not true when dealing with objects. If an object is retrieved from a list, any changes made in the object are reflected wherever the object is referenced. For instance, if aCard is a temporary variable and points at the object that is pointed at in the fifth position of hand via the statement aCard ← hand at: 5, subsequent changes to aCard are reflected in the object pointed at in the fifth position of hand.

Consider what happens if we want to group or sort the same collection of objects in different ways, perhaps for sequential access via an OrderedCollection and for direct access via a Dictionary. In this case, changes made to the objects are independent of the structures used to access them.

In standard languages, you would either have to create multiple lists, again leading to problems associated with duplication, or have to maintain multiple lists of pointers, which leads to more complex coding.

```
(a)  classList do: [ :aStudent |  (aStudent gpa >= 2.0)
                                    ifTrue: [ aStudent printName]
                                    ifFalse: [ aStudent alertInstructor]

(b)  aCard ← deck dealACard.
     hand at: 1 put: aCard.
```

Figure 4. (a) Enumeration message do: and conditionals; (b) dealing a card.

(Duplicates can be made in Smalltalk, but they are not implicit and require sending one of the built-in messages for copying objects.)

The object-message paradigm and encapsulation tend to promote a more modular system because each message represents a module. This is desirable because smaller modules are easier to create and understand. In addition, there is also a tendency for each message to be a coherent unit because the very notion of sending messages focuses the development of each method on the semantic intent of the message.

And the reason the Smalltalk programmer can focus on the intent of each message — and is not tempted to embed additional functionality in messages — is that problems of interfacing modules, which are generally associated with bottom-up development, are essentially absent when working with objects and messages. (We view method creation as bottom-up and class creation as top-down).

One reason interfacing problems are minimal is that objects are generally passed as arguments in messages, and their instance variables do not explicitly appear. Consequently, changes in an object's structure have no implications for a vast majority of the messages in which they occur.

For example, in Figure 3, wholesale changes in the structure of the class Person, and even changes in the structure of its instance variable, name (the one of primary interest in the method), will have no effect whatsoever on the method for switchNamesWith:, and no changes are required in any other method in which this message selector appears.

Why? Because objects and their instance variables are not typed, and the binding of objects to instance variables occurs at runtime. Late binding promotes fearless programming because it lets the designer postpone typing and structural decisions not germane to the current state of the prototype.

It has been our experience that many methods will not require changes or recompilation as the prototype advances. Those that require recompilation are typically small and require minimal time because compiling is incremental.

The existence of predefined object classes also contributes to this sense of fearless programming in that you have a wide selection of objects and messages to choose from, thus gaining a considerable head start in prototyping a system.

An additional advantage is that existing methods can be borrowed in a variety of ways. In some cases, the method can be directly copied to another class without change to its form or content. In other cases, the message will retain the same form — the message selectors and arguments remain unchanged, with only slight modifications required in the method. This often occurs when subclasses override messages.

Whatever problems arise and whatever morass you create tend to be at higher rather than lower levels of abstraction.

And, as indicated in the discussion of Figure 3, this can be accomplished without fear of difficulties arising from scoping or naming considerations. Names do not have to be artificially modified to distinguish messages with the same selectors but can be used with different classes and subclasses.

Perhaps the most important contribution to fearless programming, apart from the environment, is that working with objects and messages has important analogs to working at the level of human cognition. Whatever problems arise and whatever morass you create tend to be at higher rather than lower levels of abstraction.

The use of messages conveys much about the semantics of operations on objects, reducing the need for documentation. (Other documentation requirements, including managing hierarchies and messages, are handled by the environment.) A great deal of clutter, unessential low-level detail, is eliminated from much of the code. This is, in part, due to the existence of predefined classes and messages.

For instance, there are various collection classes (including Set, Bag, OrderedCollection, Array, and Dictionary) that respond to the same message, do: aBlock, to enumerate the objects of the collection and to operate on each in turn by executing the code in aBlock. Incrementing variables to process the collection is unnecessary. (Furthermore, these structures can be substituted for one another to improve performance in later stages of the prototype, often without any other changes in the code.)

In addition, objects are prepackaged bundles of parameters to be passed as arguments in messages and therefore tend to reduce the number of arguments present and to enhance the readability of the code.

Example. This example demonstrates these advantages. It is drawn from our work on relational databases but is simplified here. We will call a functional dependency a statement of the form $a \rightarrow b$ that we can read as "a implies b," as in propositional logic. It is possible that, given a collection F of functional dependencies, some dependencies are redundant.

Removing redundant dependencies from F is important in relational database design. For example, given the collection

$$F = \{ a \rightarrow b, b \rightarrow c, a \rightarrow c, b \rightarrow d, e \rightarrow c \}$$

the third functional dependency is redundant since it can be derived from the first and second using the transitivity rule:

if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

There is a straightforward algorithm to determine if a functional dependency is redundant. As an illustration, to show that the functional dependency $f = (a \rightarrow c)$ is redundant, first form $F' = F - \{f\}$ (delete f from F).

Now pass through F' as many times as necessary to discover all attributes implied by the left side of f, $\{a\}$. All implied attributes, including a, are placed in a collection called closure, the closure of a with respect to F'.

In this example, closure = $\{a,b,c,d\}$ at the end of the passes over F'. If closure contains the right side of f, then f is redundant, which is true in this example, and it is deleted from F.

A natural starting point for developing an implementation to eliminate redundant functional dependencies is to define a class called FunctionalDependency. This class can be made a subclass of Object because no other class exists for which it would be a natural subclass.

It also seems appropriate to create two instance variables to represent the left and right sides (lhs and rhs) of the dependency. The declaration is

```
Object    subclass: #FunctionalDependency
          instance variables: ' lhs rhs '
```

The methods for new, lhs:, rhs:, lhs, and rhs can easily be coded to create an instance of FunctionalDependency and to set and retrieve the values of its instance variables, respectively. We store the set of functional dependencies F in a collection named SetOfFDs.

It is quite striking just how quickly and with such little code this and other algorithms can be implemented in Smalltalk.

Structural changes. Now suppose that, after having developed the algorithm outlined above, we believe it would be more efficient to mark a functional dependency f as inactive rather than to delete it from the SetOfFDs to form F' and then reinsert it into F if it is not redundant.

This can easily be achieved by adding a new instance variable, active, to the class FunctionalDependency. No other method defined on this class nor any other code in which functional dependencies are passed as arguments or sent existing messages need be changed to accommodate the addition of the new instance variable.

Consequently, a great deal of recompiling and relinking will be avoided. The changes will be isolated to modifying the method for the algorithm. This simply involves replacing statements for removing functional dependencies from SetOfFDs to form F' by statements for setting their instance variables active to false, testing whether a functional dependency was active before using it in computing closure, and resetting active to true if the functional dependency is not redundant. At this stage, the method for eliminating redundant dependencies might look like the code in Figure 5.

The SetOfFDs in Figure 5 may be any of several predefined classes of collections in the Smalltalk system. Thus we can freely change the structure of the SetOfFDs to determine which gives the best performance. When this method is compiled, it is not necessary to have determined which class the SetOfFDs comes from because of the delayed binding.

Note that the functional dependencies in F above are not grouped by common left sides. Some algorithms require that they be grouped this way. One approach is to sort SetOfFDs according to their left sides. However, if the original set must be maintained (to allow, for example, direct access to functional dependencies if SetOfFDs is a dictionary), we can easily create collections using existing Smalltalk classes, denoted here as DepWithLHS(X), which will contain the dependencies with a common left side X.

But since we are working with functional dependencies as objects, if a functional dependency accessed from DepWithLHS(X) is made inactive, it will also be inactive if accessed from SetOfFDs. The change to the code in Figure 5 to accommodate the change in the algorithm is straightforward.

Also, we haven't yet written the method for computing closures, which is used in the second line of Figure 5 by sending the message closure to the left side of each functional dependency, closure ← eachfd lhs closure. As the code in Figure 5 is compiled, the system will notify the programmer and give an option to proceed and define the missing message later.

When developing our algorithms, we tried many approaches to computing closures. Some were conceptually similar but had different efficiencies, and others were conceptually new. In all cases, very little code had to be changed to accommodate the different versions of closure; clearly no changes were required for the code in Figure 5, and it was straightforward to borrow extensively from one version to the next.

While Smalltalk code may seem a bit strange because of unfamiliar naming conventions and syntax, it is generally true that after some initial difficulty it seems quite readable. Unessential details do not clutter the code, as Figure 5 shows. Only infrequently are lines needed to increment variables; parameters are not required for invoking routines, as they are hidden in the instance variables of the objects; and messages are simple, and their names reflect much of the semantics of what they do.

If a message is not understood by the programmer or reader of the code, it can be quickly examined. We do not mean to suggest that this cannot be accomplished in other languages, but it seems to occur more naturally in Smalltalk.

Conceptual changes. While implementing various types of closure in Smalltalk, we discovered a new approach to eliminating redundant functional dependencies. It involved deactivating all of the functional dependencies in a given DepWithLHS(X). As a result, we couldn't use the instance variable active to determine if a functional dependency was redundant.

One of the changes required to accommodate this was to introduce a new instance variable, redundant, for the class FunctionalDependency and to set its value

```
SetOfFDs    do: [:eachfd | eachfd active: false.
                           closure ← (eachfd  lhs) closure.
                           (closure includes: (eachfd  rhs))
                             ifFalse:[each fd active: true]].
```

Figure 5. Eliminating redundant dependencies.

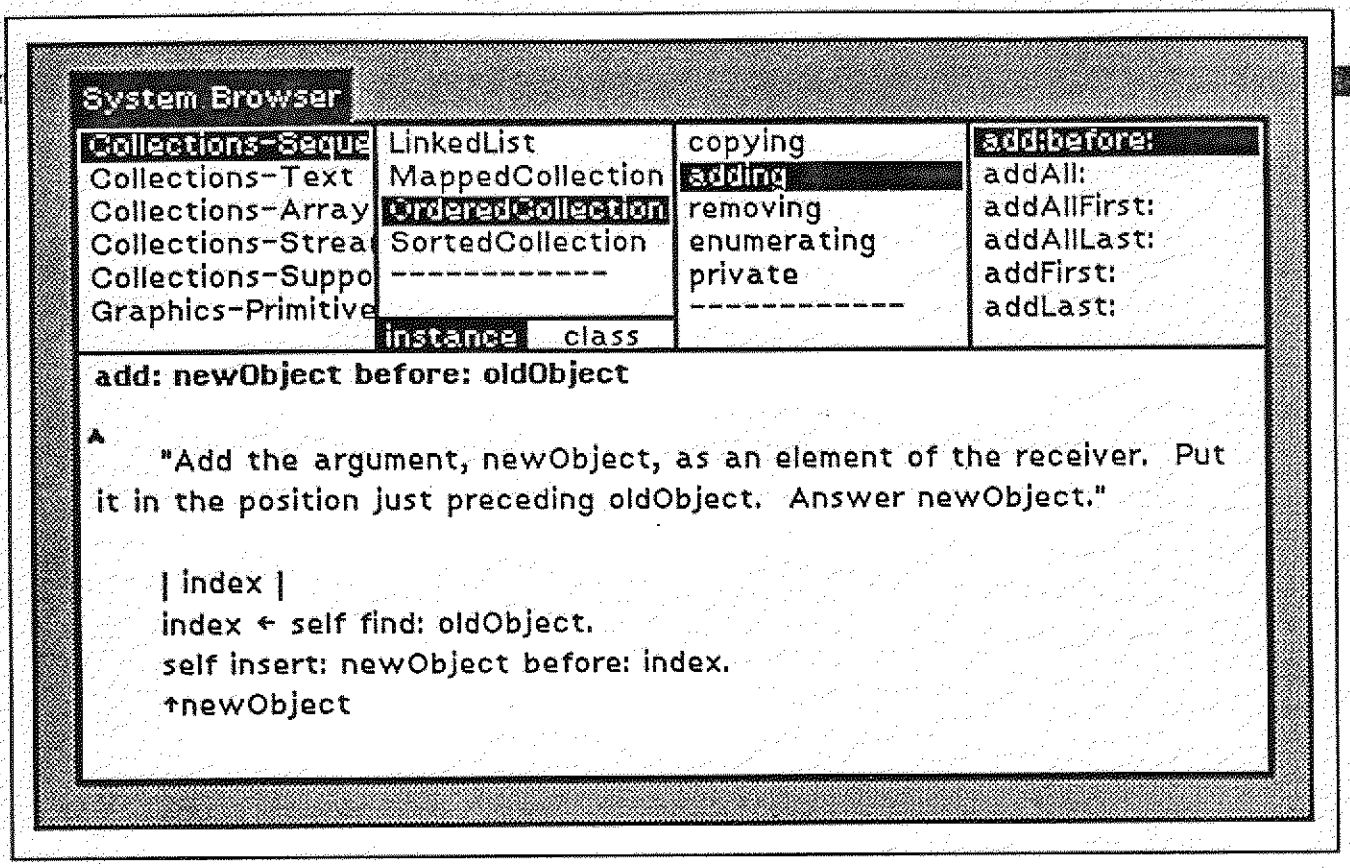


Figure 6. The system browser.

to true when a given functional dependency was discovered to be redundant. (For several reasons, we chose to save redundant dependencies rather than discard them.)

Though this is a small change, it nevertheless hints at how changes made at the conceptual level can be translated directly into messages and objects. This shows that the way we think about entities in an application can be translated directly into two fundamental aspects of an object, its attributes and its behavior. Of course, other parts of the code in other methods were affected by changing the semantics of the instance variable active. The Smalltalk environment provides the tools to work rapidly through the effects of these changes.

The Smalltalk system promotes trying alternatives. The time required to restructure objects, modify methods, create new methods, locate the effects of changes, and recompile the code to experiment with it, is much less of a factor in the cost/benefit ratio than you would incur if working with most standard languages.

Also, most changes occur at a high conceptual level, so it is more like working with changes in the specification than with changes in the code. However, we do not claim that Smalltalk is close to achieving the software engineering goal of directly compiling specifications.

Environment

While encapsulation and hierarchy form the basic foundations of the Smalltalk language, it is the rich environment that lets you work with its many classes, methods, and messages.

In Smalltalk, there are several window types available. These windows can be created, moved, reshaped, collapsed, and closed using a three-button mouse (some systems use a single-button mouse). Multiple, overlapping windows can be on the screen, and you can move from one window to the next to carry out different or related tasks.

Code can be modified and run from different kinds of windows, and application windows can be activated and deactivated. This is particularly useful if an error turns up while debugging an application and recovery from within the application is not possible.

The convenience of doing different things in the system quickly and efficiently also contributes significantly to fearless programming. You aren't caught up in the time-consuming cycle of doing something in edit mode, exiting and entering compile mode, exiting and entering run/debug mode, exiting and returning to edit mode to make changes. Tesler[4] has discussed the philosophy behind modeless environments and the early Smalltalk interface.

Organization. Because there is no single linear command list, beginners often wonder where the program is. Programming in Smalltalk is mainly adding new classes and messages, creating objects, and passing messages. Thus, being able to move around the system easily and to work with individual classes is critical to productive Smalltalk programming. The system classes are organized in a hierarchy with the class Object at the top, but you don't have to remember the exact hierarchy while programming because the interface provides a convenient organization method and access to the system classes through the system browser, shown in Figure 6.

The system browser window contains several panes, each with its own menu of actions. The second pane from the left along the top contains the name of classes, and the fourth pane contains message selectors. The first and third panes catego-

rize similar types of classes and messages, respectively.

In Figure 6, the category of classes selected (indicated by reverse video) is Collections-Sequenceable. This class category contains classes linearly structured. One such class, OrderedCollection, is selected in the second pane.

The instance/class pair at the bottom of the second pane is a toggle. If instance is selected (as shown), what appears in the panes to the right and below will pertain to methods sent to instances of the class. If class is selected, it will pertain to methods sent to the class, which is also considered an object.

The various categories of messages that can be sent to instances of OrderedCollection are shown in the third pane. Here, we have selected messages for adding objects to an ordered collection. The actual messages for adding are in the fourth pane with the message add: before: selected. The method for this message is shown in the large pane at the bottom of the browser window in Figure 6.

Using the browser. For the most part, the browser is used to create classes, create messages and methods, and to browse through the system. To indicate how these tools can be used to make changes, recall the example of working with the class FunctionalDependencies. There was a change in the semantics of the instance variable active when the instance variable redundant was added to the class. This required examining all the methods that might be affected, so we want to review all methods that modify or retrieve the instance variable active.

There are several ways to do this. One way is to select the class FunctionalDependency in the browser. We can then get a pop-up menu (Figure 7a) and select the item, inst var refs. Another pop-up menu appears (Figure 7b) that lists the instance variables for the class FunctionalDependency. By selecting the variable active, we get a window showing all methods where this instance variable appears (Figure 7c).

By examining each method, we see that the message active: modifies the value of the instance variable active and the message active retrieves it. After selecting the method for active:, we can get a pop-up menu in the same window (we don't have to return to the browser) from which we can select an item, senders (see Figure 7d).

In this case, we are asking for all methods that use (send) the message active: in their code; the result is the window shown in Figure 8. By selecting each method, we can examine the code to see if the change in active's semantics requires changes in the method. Whatever the changes may be, we can make them in this window (its top part is shown in Figure 8) and compile the code there.

Three items in the menu in Figure 7d — senders, implementers, and messages — can be very effective message tracers. Again, messages with the same name (but different functionality) might be imple-

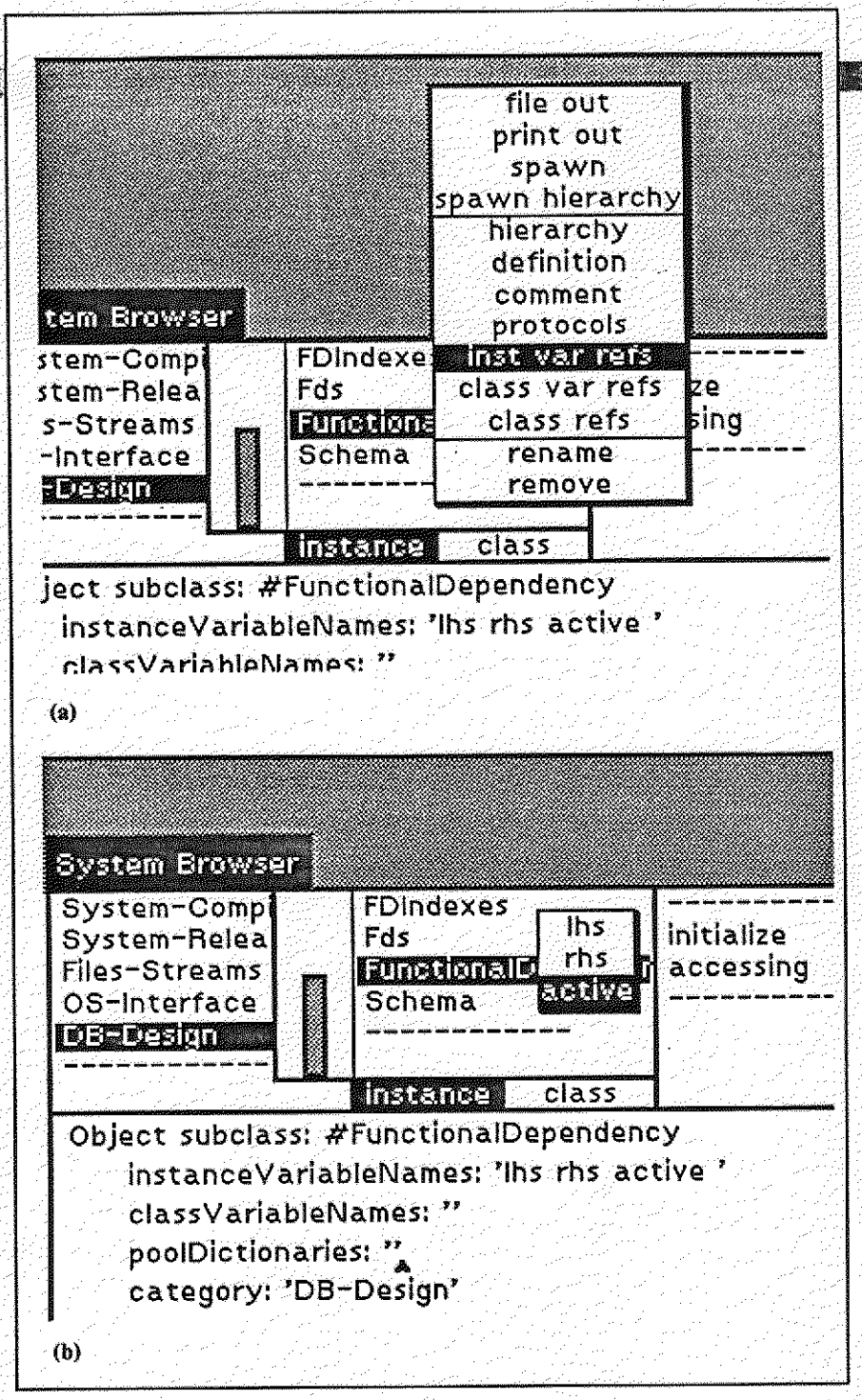


Figure 7. (a) Class pane menu; (b) instance variables.

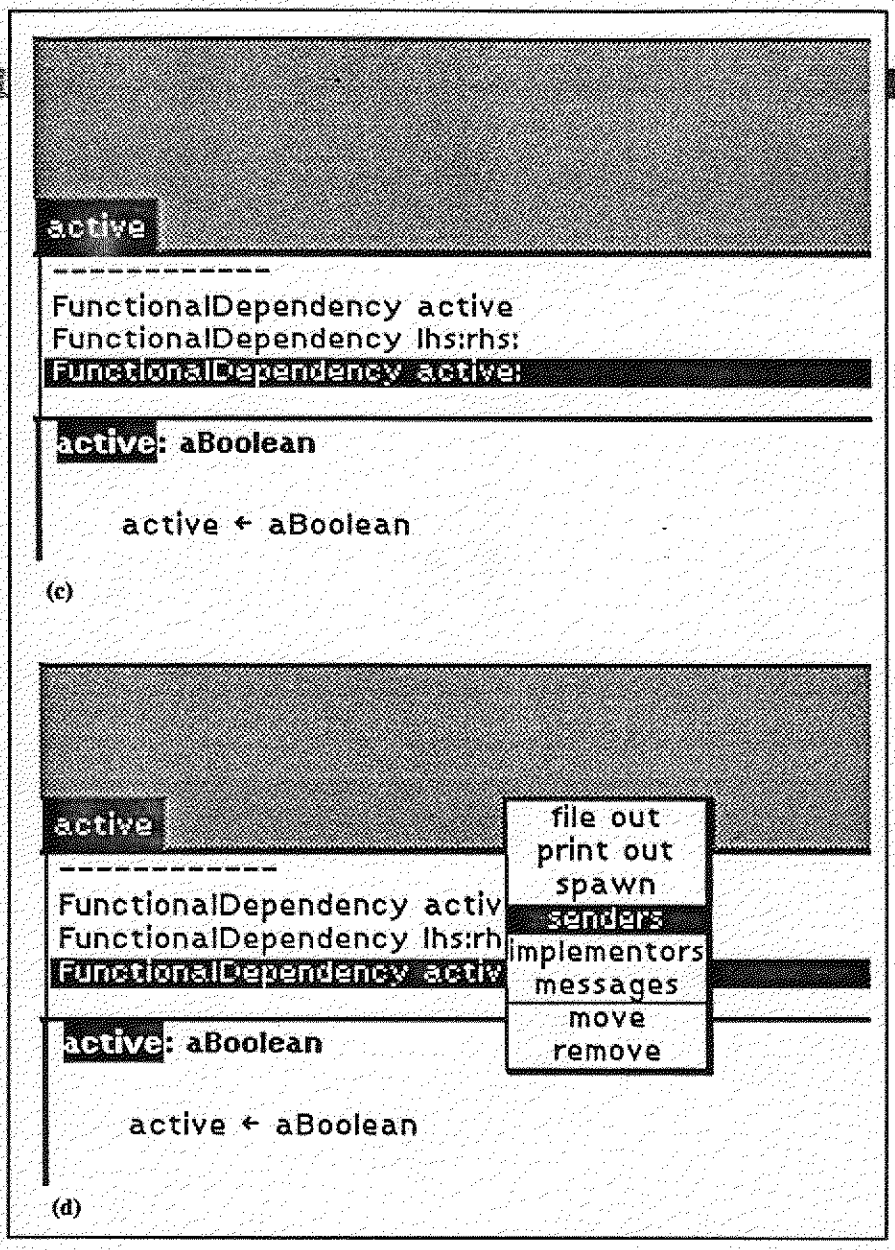


Figure 7. (c) Methods using active; (d) finding senders of active.

mented in different classes in the system, and a given method might use messages from other classes. The item, messages, can be selected directly from the menu in Figure 7d; it is also a menu option for the top pane in Figure 8. Selecting it will quickly get a menu of all messages in the selected method, as illustrated in Figure 9. If we select any of the messages in the menu in Figure 9, we will get another window of the type shown in Figure 8, which lists all classes where a message of that name is implemented. We can now select one class and inspect its method in the lower pane of the new window.

These facilities help the user browse through the system, as well as debug. In addition, the system has utilities to identify and correct syntactically and semantically incorrect code. The power of the complete tool collection contributes significantly to fearless programming.

Smalltalk source code is compiled into an intermediate form, called bytecodes, that is then interpreted. This compilation is done incrementally as new classes, messages, and the associated methods are defined. Code is displayed in the code pane of the browser and is compiled through the menu for that pane.

If the code contains a syntax error, an appropriate message is inserted into the code in reverse video. Because the message is in reverse video, it can easily be cut, the code can be corrected, and the method can be recompiled.

For browsing, a selection in the system browser's code pane, explain, is very helpful. You can select any token in a method, and the explain selection will indicate what type of token it is (class variable, global variable, name of a message, and so on) by inserting explanatory text in reverse video in the method display in the code pane.

Debugging. While changing the semantics of the instance variable active and introducing the new instance variable redundant, suppose we failed to initialize redundant to false when creating new instances of FunctionalDependency. This would lead to problems, which can be traced with the built-in debugger.

One approach is to place a halt in the code where the problems arise (in this case, the method for eliminating redundant dependencies) and to step through the code and examine the objects. When code is executed, a notifier will appear on the screen to indicate that a halt has been encountered. You have the option to proceed past the halt or to enter debug mode.

If you choose to debug, the window shown in Figure 10 appears, and the menu appearing over the top pane can be obtained. The top pane of the window contains the current activation stack — the list of messages leading to the halt. This list can be scrolled, and individual messages can be selected. Code for the selected message is displayed in the middle pane, and the bottom two entries in the top pane's pop-up menu can be used to run through a stepwise simulation of the program.

Selecting the item, step, executes the current selected message (redundant in Figure 10) and moves you to the next one (ifFalse:). Selecting the item, send, enters the method for the selected message, which then displays in the middle pane and can be treated the same way. The lower left pane contains instance variables of the message receiver displayed in the middle pane, and the lower right pane contains current values of all temporary variables in that method. The value of the loop variable, each, is an instance of class FunctionalDependency at this point.

You can inspect this object, as Figure 11 shows. The result is the window in Figure 12, where we have selected redundant,

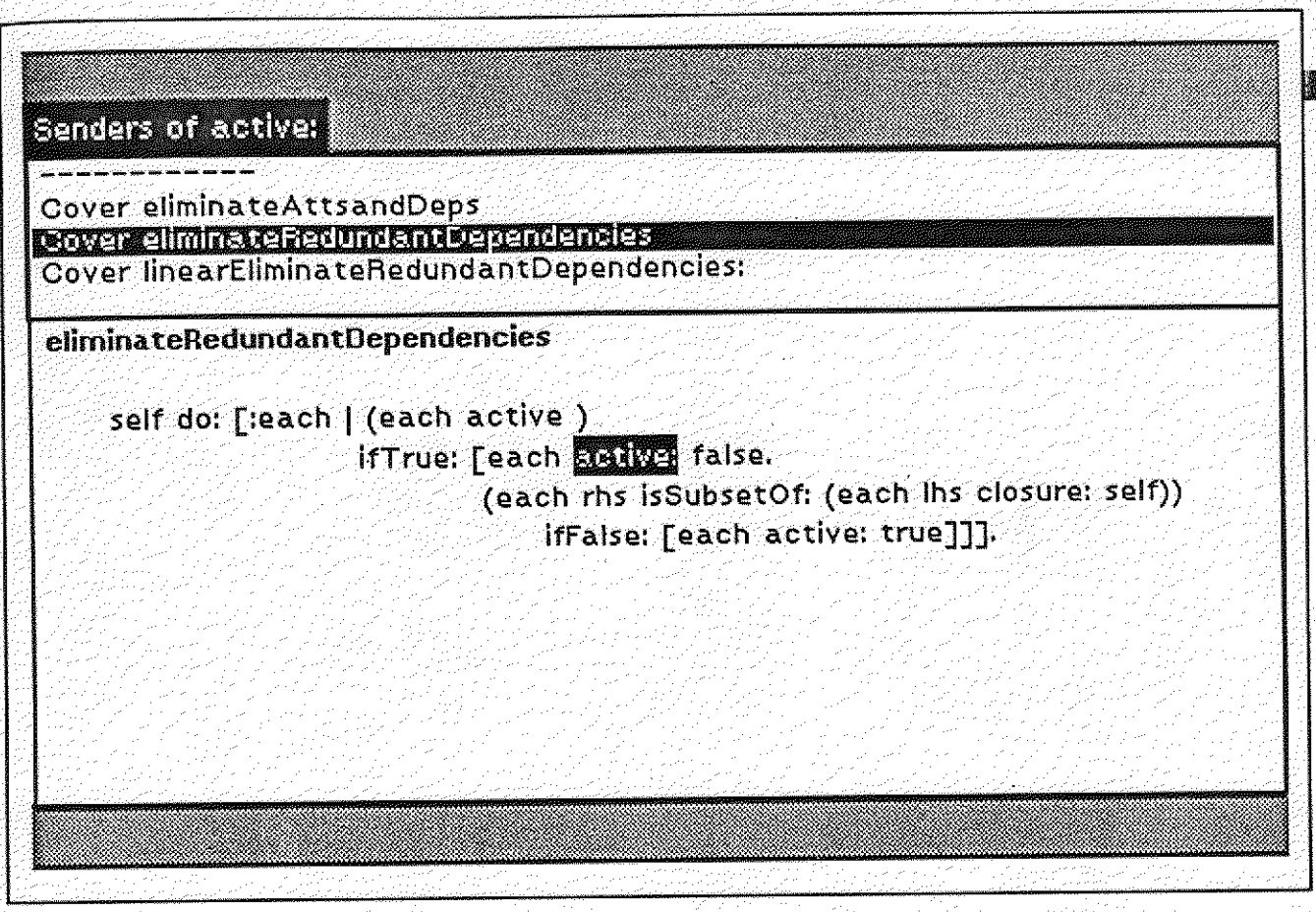


Figure 8. Senders of active:.

whose nil value shows in the right pane and reveals the problem that redundant was not initialized. Chapter 19 of the Orange Book[5] gives a more detailed description of the debugger.

There are other ways to invoke the debugger. For example, if a halt had not been placed in the code, a notifier would appear during execution upon test of the nil value of redundant. The notifier would indicate that a non-Boolean receiver had been encountered, and three options would be available:

(1) Set the value of the receiver to true and proceed with execution.

(2) Enter the debugger at the point of the problematic code.

(3) Stop the errant process simply by closing the debug window.

In the second case, the debugger window has the form of the one in Figure 10, and the programmer can proceed as above. The debugger is a particularly nice tool and certainly helps make runtime errors much more tolerable during program development. It can be very confusing to the beginner, however, as the depths of the code are explored with the option send. For example, a user who doesn't know how system-

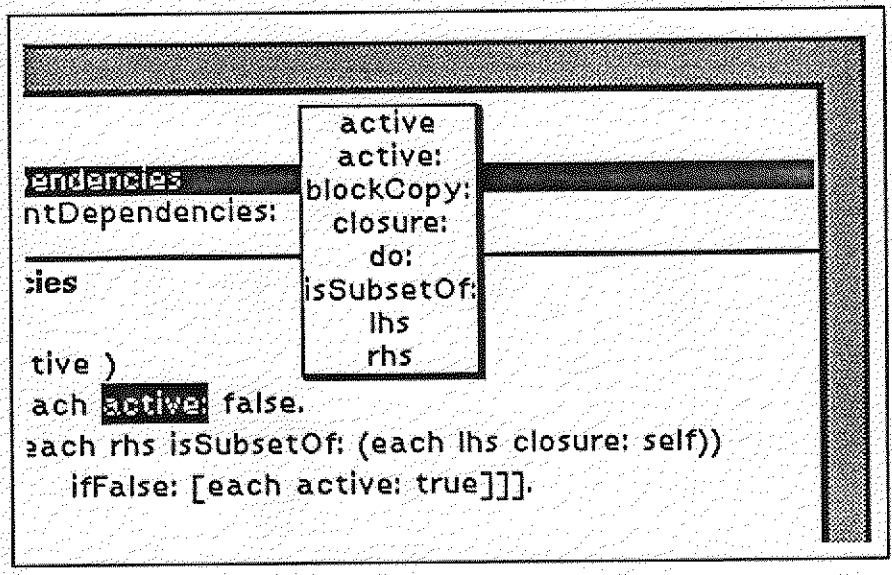


Figure 9. Messages in a method.

defined control structures are implemented could be confused by the system code when it is encountered.

Modifying Smalltalk. Fearless programming gives users confidence to attempt extensive changes in the system, even ones to the predefined classes. An example of major restructuring from our database project would be too detailed to develop here, but we can give an example that has pervasive implications for the system.

We needed various ways of writing bitmaps of parts of the screen out to files, and the system did not already have all the necessary functionality. To capture bit-

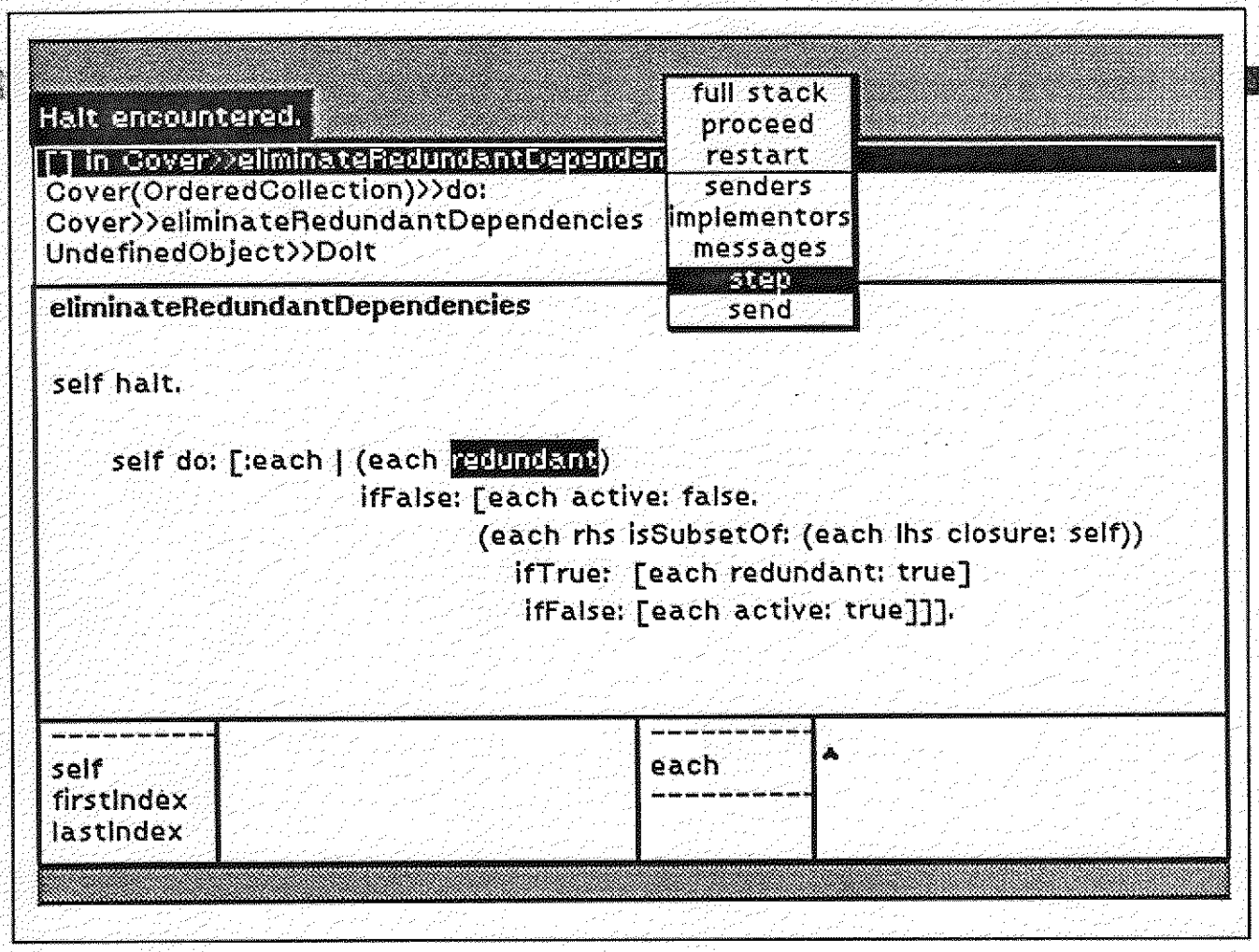


Figure 10. The debugger.

maps with pop-up menus displayed at the time of capture, we had to deactivate the menus so they could be brought up on screen to have their picture taken without having the selected message sent when the button was released.

To achieve this, we added a class variable, Active, to the class PopUpMenu, which governs pop-up menus, through which essentially all actions are initiated. If Active is true, all actions through menus will proceed as usual; if Active is false, a conditional placed in the code for sending menu messages returns a no-op value when the mouse button is released, and the menu disappears with no action taken. We then defined a two-key sequence to toggle Active.

This change is very powerful because it covers the predominant means by which actions are taken in the system. This also makes it dangerous. It indeed led to several interesting debugging problems, two of which we will illustrate.

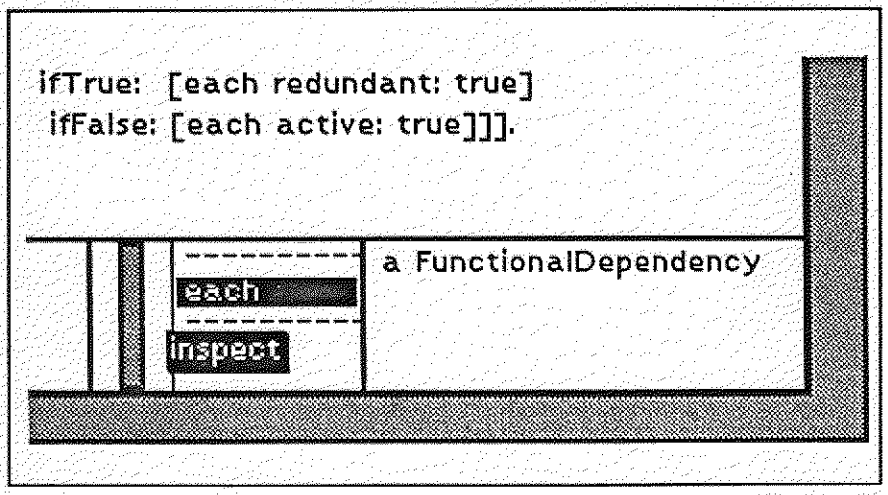


Figure 11. Inspecting a temporary variable.

Fatal error. If the class variable Active is defined, but initialization to true or false is neglected, its default value will be nil. If anything is then executed through a pop-up menu, this nil value for Active will be encountered in the conditional code, and a notifier will appear. The menu for this notifier contains two options, proceed and debug, which would ordinarily function as described above.

In this case, however, using the middle button to generate the pop-up menu from which to choose an option would result in consultation of the same conditional code, which would again find that Active has the

value nil. This would generate another notifier, which in turn would generate another notifier if we selected proceed or debug, and so on.

An attempt to close this window with the mouse and thus stop the process would also generate a notifier. The toggle mechanism could not now be used to set Active because that mechanism uses a conditional based on Active being true or false. Furthermore, you cannot shut down the system because a pop-up menu is also required to do so. The only choice in this case would be to abort the system itself.

To guard against major damage under such a condition, in fearless programming you take snapshots of the entire environment from time to time. After an abort, the system can immediately be recovered at the last snapshot, and all changes in the interim can be recovered from the changes file.

Avoiding a system abort. Our initial attempt to name the bitmaps we were capturing illustrates a serious problem that did not require a system abort. The capture was accomplished by executing code in a code pane of a window that produced a small window to type the name of the file to store the bitmap in, provided the value of Active is false. Unfortunately, when this code is executed, the naming window that automatically pops up uses an instance of PopUpMenu, which also finds that Active is false, and which therefore produces a new window asking that the naming window be named.

Unlike the previous example, the entire system functionality was not lost and there was a more elegant solution than a system abort. We can toggle the value of Active to true with the appropriate key, activate a system browser with a single click of the mouse, find the offending method in class PopUpMenu, change the code by deleting the conditional on Active, and recompile the method. When we return to the running application it will use the new, incrementally compiled code and stop asking for naming windows.

There will of course be several stray windows on the screen that will have to be closed, and there are various conditions in different versions of Smalltalk that might prevent extrication from such a predicament without aborting the image. The permanent fix to this problem was relatively straightforward. The main point is that we changed the system during an application run when our modifications cut too deep — and this ability contributes to the sense of fearless programming.

In fearless programming you take snapshots of the entire environment from time to time.

The environment in general — not just the debugger — has the tools needed to build, examine, modify, and test code. These tools are readily available and not limited or constrained by mode or context. Building, examining, modifying, and testing code are all related operations. While using the debugger, for instance, you aren't trapped in it. You can browse, modify code, and carry out a variety of tasks.

One measure of the Smalltalk environment's effectiveness is the extent to which hard copy is unnecessary during application development. Initially you use hard copy to sketch ideas and code, but as development proceeds your work becomes virtually paperless. The only reason for printing out code is to carry it away from the machine or for backup. Because we have always had to use printouts of various iterations of the code in other languages, we believe this to be a strong indication of the Smalltalk facilities' effectiveness.

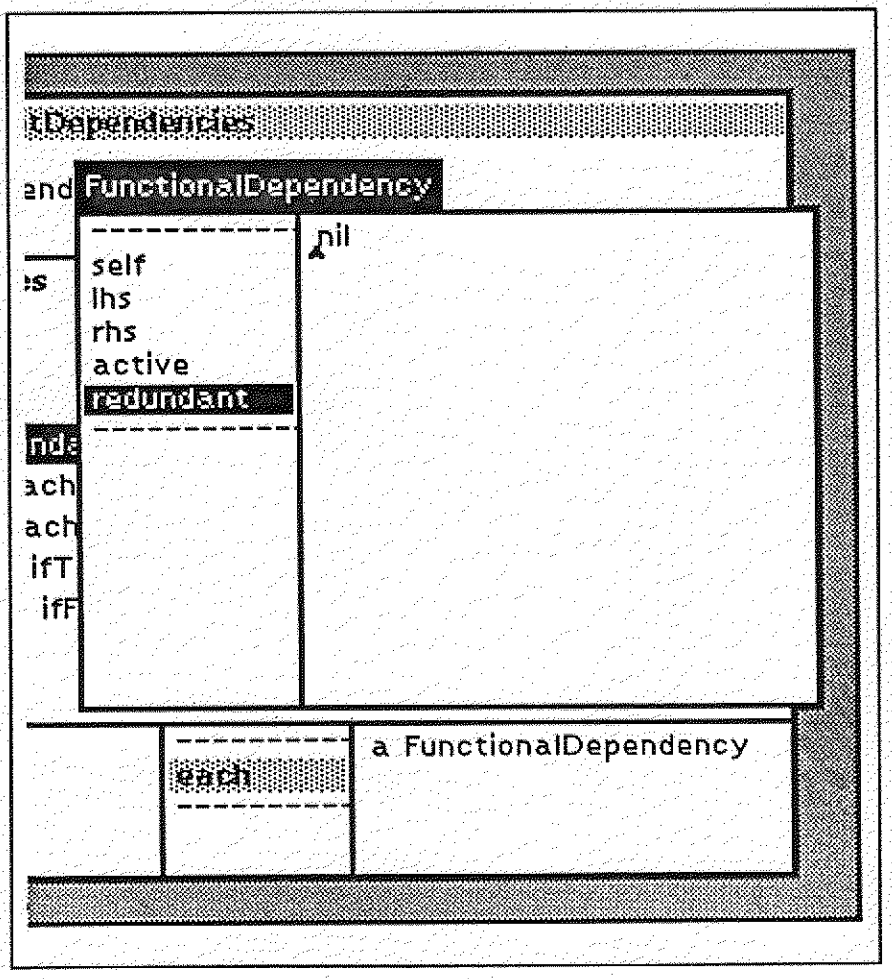


Figure 12. Viewing an object.

Other window types. In addition to the system browser and debugger windows, there are several other important window types. Code written in a workspace is not compiled into the system, so the workspace is generally used to try out existing messages, run applications, and develop and test new methods. A file list window is used to interact with the underlying file system.

Some versions of Smalltalk have windows to interact with modules written in other languages. Other windows, called system transcripts, let the system note that actions have been accomplished or special conditions have been encountered. Another window, called a system workspace, contains commonly needed code segments. A template for such segments can be modified in the system workspace and be executed on the spot.

For example, we might know that some class implements the message, suspend, but not know which one. We can activate a system workspace and locate the template

```
Smalltalk browseAllImplementorsOf:
#keywordSymbol
```

This can be easily edited to

```
Smalltalk browseAllImplementersOf:
#suspend
```

It can then be selected and executed, and the result will be a window such as that shown in Figure 8, which displays all methods that implement the message denoted by suspend.

Although Smalltalk provides excellent facilities to browse the structure of classes and objects and to examine their messages, it does not provide adequate facilities to examine the behavior of a complex application. Cunningham and Beck[6] have reported one effort to remedy this.

Misconceptions

Three Smalltalk areas are often misunderstood.

Performance. It is commonly believed that the abundance of high-level features, the uniform object-message implementation, the delayed binding of variable types, and the graphical capabilities of Smalltalk yield a system with poor overall performance.

In part, this can be attributed to the fact that the initial prerelease copies distributed by Xerox PARC to participating companies for refinement in the early 1980s were quite slow on different hardware implementations. It also did not help the performance image of Smalltalk that a major implementation on a VAX was particularly slow in a time-shared environment, and that the project thus limited its emphasis to implementations on workstations.[7] And it is well known that an absence of data typing and the lack of a global optimizing compiler typically exact heavy performance penalties.

We have found that there are nonenhanced commercial versions of the language that are quite fast.

Research is now under way to add performance enhancements to Smalltalk, such as improved garbage collection, faster alternatives to the bytecode interpreter, and typing facilities. With a data-typing facility, you could delay all binding during prototyping and product development for maximum flexibility and then identify critical sections and optimize, using data typing and other methods of fine tuning, for the mature application.

We have found, however, that even without such enhancements, there are commercial versions of the language that are quite fast. In particular, we implemented and tested database normalization algorithms in MProlog and Smalltalk on a Tektronix 4404 and found Smalltalk to be about 15 times faster than MProlog. After performance tuning in both languages, which was far more straightforward in Smalltalk, the gap between the two languages roughly quadrupled.

Just in case we simply had a particularly slow implementation of MProlog, we implemented and tested one set of procedures to find graphical shapes in points in the plane in C, Smalltalk, and MProlog — and the results were striking. The MProlog program was about 80 times slower than the C program, but the Smalltalk program was a bit less than two times slower than the C program. The only performance enhancements we made were to correct obvious inefficiencies in the code.

For example, the Prolog program as originally published[8] was 500 times slower than the C program, and with a few rather obvious inefficiencies made for a rather unfair comparison. Some twiddling of the Prolog code eliminated a large amount of unnecessary backtracking and lowered the ratio to 80. We also ran the C program on a VAX 11/750, and the runtimes were within 15 percent of the runtimes on our workstation.

These limited tests certainly cannot even begin to characterize the performance of Smalltalk relative to MProlog and C, but they underscore what we had discovered: On our workstations, Smalltalk is very responsive during program development, and it performs quite well on the type of computation and data manipulation we are using.

The performance standard, which different versions of Smalltalk on different machines are measured by, is the language's performance on the Xerox Dorado, on which it is very fast. The Dorado is a descendant of the Xerox Alto and is a high-performance experimental microprogrammed personal computer with a microcycle time three times faster than the VAX 11/780.

There are commercially available workstations that run special Smalltalk as fast or faster than the Dorado. This performance is excellent and makes for an outstanding software prototyping environment. We would be willing to pay a far bigger performance penalty than we now pay in a language like C, for example, for the productivity gains we have achieved in Smalltalk.

Smalltalk provides system facilities to trace execution at varying degrees of granularity, to time code blocks, and to identify critical sections. There is a class named Benchmark that contains methods for micro and macro benchmarks to measure the relative efficiency of different bytecode interpreters.

Applications. Many commercial advertisements for Smalltalk characterize it as an artificial intelligence language. Indeed it may be excellent for a wide variety of artificial intelligence applications, and we are using it to create a production system for database design. However, it seems to be far more general-purpose, and it is not generally considered an artificial intelligence language by the artificial intelligence community.

The object-message paradigm provides a powerful general-purpose programming language. The graphical capabilities provide high-level primitives not only for the development of interfaces but for domains that need to use graphics. Built-in classes provide an excellent foundation for many applications.

A good example of built-in capabilities for other purposes is a set of classes that support discrete-event simulation. As a descendant of Simula, Smalltalk contains the basic mechanisms to support quasi-parallel processing, through the classes Process, ProcessorScheduler, and Semaphore. These allow process description and referencing, dynamically generated processes, and delimited and sequenced active phases of processes, with or without reference to the concept of system time.

Important capabilities are scheduling and executing processes of different priorities and easily suspending a process and resuming it later. Higher priority processes are executed before lower ones, and events with the same priority are handled on a first-come, first-served basis.

Part three of the so-called Blue Book[9] Smalltalk reference is devoted to discrete-event simulation. It develops the basic classes in addition to the built-in ones, and it develops several of the applications in Birtwistle's book[10] sufficiently to provide an excellent basis for simulation.

There are numerous significant errors in the simulation code in part three of the "corrected" first edition of the Blue Book, but these are being repaired and some of the code has been streamlined. One disadvantage of working with the simulation classes is that the debugger does not function well while tracing the suspension and resumption of processes. A simulation trace can be started in the debugger, but counts of active processes start to degrade, and certain debugger windows are not functional without coercion (such as hitting the abort key). For debugging complex and lengthy processes, the user cannot rely on the debugger.

Graphical paradigm. The basic paradigm for handling graphical applications in Smalltalk is through the model-view-controller triad. For the display of an object, the object itself is considered the model, the graphical layout is the view, and the coordination of user inputs to examine the model and the display is done through the controller.

Simple and complex nested views may be created and manipulated. Basic classes handle generic views and controllers, and the system interface itself is handled with the MVC, providing a rich set of tools serving as models for the programmer's development. Unfortunately, the promised and long-awaited tome from Xerox PARC on how to use the MVC has not been published, which leaves a hole in the documentation.

Coupled with the richness of the system, this makes learning to use the MVC a formidable task. Like the overall Smalltalk system itself, the MVC is rich and powerful, and this richness contributes to the initial learning difficulties.

Apparently grumbling about using the MVC is relatively common, and it is rumored that an alternative is sought. The concept of a pluggable view removes the need to build each application from scratch, but it certainly does not represent a radical departure from the MVC. Indeed, the Apple version of Smalltalk uses another Apple product, MacApp, heavily to build user interfaces in Smalltalk — a departure from using the MVC.

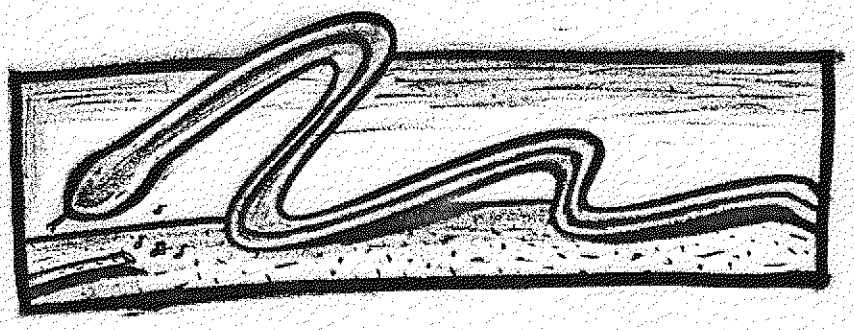
Thus, while Smalltalk contains a well-integrated set of graphical primitives, the beginning Smalltalk programmer should not expect to find a quick and easy path to developing graphical applications.

Learning experience

Smalltalk is not easy to learn, even if you have considerable experience with standard languages. The Blue Book and the Orange Book are excellent references, especially when coupled with the extensive on-line documentation — but they are not good textbooks.

The Blue Book deals mainly with features of the language, while the Orange Book deals with the environment. But the system is so rich and so well integrated that it is not easy to decompose the task of learning it into subtasks. It is like the classic chicken-and-egg situation — the rich set of tools can be used to overcome the complexity of the system, but at the same time you have to work with the system for some time before seeing how the tools can make a significant difference in working with the system.

We think learning Smalltalk is not like learning a language but like learning a culture. Over time, improvements in methods of learning the system should emerge, but the current paradigm involves intensive reading of both reference books, browsing the system on line, writing your own applications, and frequently moving from one reference or source to another. Fortunately, learning the system poses only a short-term disadvantage, but long-term productivity gains may be worth the initial investment.



Smalltalk promotes high productivity and reasonably low rework. While this may serve as a general characterization of fearless programming, there are some specific characteristics to consider in determining the extent to which a system promotes it. Briefly, fearless programming is:

- Referenceless. The system manages pointers to structures (objects in Smalltalk).
- Clutterless. The system abstracts out low-level detail.
- Typeless. Types need not be declared, so design decisions can be made flexibly.
- Modeless. Actions can be taken at will within the programming environment.
- Paperless. The system manages documentation, and the programmer does not rely on hard copy to examine the work.

Certainly for experimenting with algorithms, for rapid prototyping systems, and for developing programs that need highly interactive and graphical interfaces, you can expect a net gain in return for the investment of learning Smalltalk. One of the major advantages is that the object-message paradigm greatly helps conquer system complexity.

While the researcher interested in straightforward number-crunching may have little to gain in using Smalltalk, and while there may be other classes of users who find other languages and development environments much more suitable, we feel that programmers in many application areas would be well-served by the Smalltalk environment.

Moreover, the good performance of some commercial systems and the enhancements of Smalltalk under development suggest that Smalltalk may even have a future niche in scientific and real-time systems as well. Smalltalk promotes a good, new approach to programming. □

Acknowledgments

We'd like to express our appreciation for the many helpful comments by the referees.

References

1. Glenn Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, Mass., 1983.
2. Jim Diederich and Jack Milton, "Oddessy: An Object-Oriented Database Design System," *Proc. Third Int'l Conf. Data Engineering*, Computer Society Press, Los Alamitos, Calif., 1987.
3. Setrag Khoshafian and George Copeland, "Object Identity," *Proc. First Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications*, ACM, New York, 1986.
4. Larry Tesler, "The Smalltalk Environment," *Byte*, Aug. 1981, pp. 90-147.
5. Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.
6. Ward Cunningham and Kent Beck, "A Diagram for Object-Oriented Programs, *Proc. First Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications*, ACM, New York, 1986.
7. David Patterson, "Smalltalk on a VAX," *Smalltalk-80 Newsletter*, Feb. 1984, pp 3-4 (available from Xerox Corp., Palo Alto, Calif.).
8. P.S.G. Swinson, "Prescriptive to Descriptive Programming: A Way Ahead for CAAD," *Proc. Logic Programming Workshop*, Architecture Dept., Univ. of Edinburgh, Scotland, 1980.
9. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.
10. Graham Birtwistle, *A System for Discrete Event Modeling on Simula*, Macmillan, London, 1979.

Jim Diederich is an associate professor of mathematics at the University of California at Davis. His research interests include database design and object-oriented systems.

Diederich received a PhD in mathematics from the University of California at Riverside. His a member of ACM and the Computer Society of the IEEE.

Jack Milton is an associate professor of mathematics at the University of California at Davis. He is also an associate investigator on the Knowledge-Based Management Systems project at Stanford University and coordinates the Database Research Seminar there. His research interests include database design and object-oriented systems.

Milton did his undergraduate work at Swarthmore College and received a master's and a PhD in mathematics from Duke University. He is a member of ACM and the Computer Society of the IEEE.

The authors can be contacted at Math Dept., University of California, Davis, CA 95616.

FEATURE

Smalltalk Yesterday, Today, and Tomorrow

A look back and a look ahead at this innovative programming language—first featured 10 years ago in BYTE

L. PETER DEUTSCH AND ADELE GOLDBERG

It's been a decade since the August 1981 issue of BYTE was published. That issue provided many people with a first comprehensive look at the then-fabled Smalltalk programming environment. In this article, we look back at how people thought about Smalltalk in those days. Then we'll look more broadly at how Smalltalk and object-oriented software technology has progressed since then; we'll also consider today's state of this technology and the market for it. Finally, we'll look ahead to objects in the year 2001, another decade hence.

1981: Sending Up the Balloon

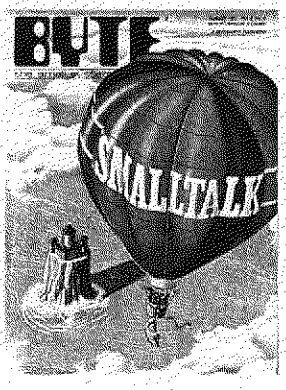
In that BYTE issue of 10 years ago, we wanted to convey three ideas about Smalltalk and object-oriented software technology: first, that an interactive, incremental approach to software development can produce qualitative and quantitative improvements in productivity; second, that software should be designed in units that are as autonomous as possible; and third, that developing software should be thought of in terms of building systems, rather than as black-box applications. The Smalltalk-80 system described in that issue so long ago was the exemplar of these three ideas.

Smalltalk was widely known then—and yet, largely unknown. Alan Kay and others from the Xerox Palo Alto Research Center (PARC) had been giving talks with tantalizing glimpses of the technology, but few people knew or understood its content. Thus, the cover of BYTE's Smalltalk issue—depicting a brightly colored Smalltalk hot-air balloon leaving an isolated island—symbolized our feeling that the time had arrived to start publicizing what we'd been doing. We believed we had new ideas that could make a real difference in how people developed software.

Many research examples developed at PARC demonstrated that object-oriented design could produce an appealing, intuitive, and direct mapping between objects in the real world and objects in a software implementation. We saw this as a radical breakthrough in one of the most difficult and problem-prone steps in software development—identifying terms and relationships as understood by human participants of a particular situation with those understood by a computer.

We believed that this simple mapping of nouns to objects was all (or most) of the story about how to design with objects, and we presented it as such in the 1981

ILLUSTRATION: PEGGI ROBERTS © 1991



The motivation of the past decade was to move Smalltalk off its island.

BYTE articles. Subsequently, in examples given in our books in 1983, we demonstrated that the power of objects applied to more than nouns: It also applied to events and processes. But this power was not as well explained or exploited.

The Smalltalk research project was founded on the belief that computer technologies are the key to improving communications channels between people, in business as well as personal settings. Our activities focused on finding new ways to organize information stored in a computer and to allow more direct access and manipulation of this information.

The Smalltalk edition of BYTE introduced our approach to managing the complex information world of modern applications. It explained our methods for taking full advantage of new graphics and distributed computing and for improving the ability of experts in business and personal computing to describe their world models.

In retrospect, we are pleased that much of the software community has come to agree that the object-oriented approach to software organization is a new way to solve problems that is often better than the procedural approach. Although our ideas about problem-to-implementation mapping were incomplete—notably given the lack of formal methodologies—those ideas are widely accepted today.

1991: A Decade of Experience

What have we learned in the past decade based on the Smalltalk research and experience that was introduced to the public in those 1981 BYTE articles? The first idea, as we stated earlier, is simply that a highly interactive, highly incremental software development environment can produce a qualitative improvement in software development productivity. Even in 1981, Smalltalk systems were not the only ones with this characteristic—Lisp systems pioneered the approach in the early 1960s—but they were among the outstanding examples and were the ones that moved most successfully from proprietary hardware to the microprocessor mainstream. Today, the truth of this idea is widely recognized: The suppliers of environments for more-established languages like C, C++, and Ada are now aiming to provide the benefits that Smalltalk introduced a decade ago.

BYTE ACTION SUMMARY

When BYTE first broke the news about Smalltalk to the world, there were no PC versions of the language. Now, the principles that Smalltalk pioneered have permeated the microcomputer world, and powerful versions of the language are available for a variety of personal computer platforms.

The second idea is the basic idea of object-oriented software organization: that software should be designed in units that are as autonomous as possible, should correspond to identifiable entities in the problem domain whenever possible, and should communicate through identified interfaces. This idea grows out of work on modular software design that dates back, again, to the 1960s. Object-oriented terminology adds an emphasis on direct mapping of concepts in the problem domain to software units, the idea of shared behavior and multiply instantiated state, and a focus on the interfaces between the units.

The last of these (the interfaces between the software units) makes it easy to think about systems that are configured or that grow dynamically. Smalltalk has no monopoly on new concepts, but it has been a leader in the public relations necessary to get these concepts out into the computing mainstream.

Object-oriented software organization has a natural relation to two current trends in software construction: combinable applications and open systems. Our interpretation of the term *open systems* is that for systems to grow, evolve, and combine gracefully, they should be constructed out of software with published interfaces. Functional software should be designed to be used as a component by other software, as opposed to being monolithically united with a particular interface designed only for humans at a terminal.

The third important idea that has grown partly out of the Smalltalk work is related to the open-systems idea—namely, that one should always think about building software in the context of building systems, rather than in the context of black-box applications. In other words, one should examine explicitly the nature of both the downward interfaces (the resources or facilities the software uses) and the upward interfaces (the client's use of the software) and make them as undemanding as possible. Separating functionality from the user interface, which is the Smalltalk concept of model-presentation-interaction known as *model-view-controller*, is just one application of this principle—but a very important one.

The motivation behind much of the activity in the past decade was to move Smalltalk off its island and into easy availability for the general programming community. We look at this activity as being aimed at creating a credible, concrete, and robust realization of the ideas that we could present only in sheltered research form in 1981.

As Smalltalk has moved into the commercial world, it has encountered the familiar phenomenon of technological life span. A technology comes into existence on paper, often at a university. It then progresses to research papers, research prototypes, and usable research-scale artifacts. Finally, it goes into commercial use, first by the adventurous and then by the broad mass of users—getting adapted, extended, patched, and transported as long as it continues to solve problems well, and eventually getting replaced in many or all of its uses by newer technology. Smalltalk is now in this third stage—past the scrutiny of the adventurous and experiencing wider commercial adoption. [Editor's note: *For a look at some new products that should help bring Smalltalk to a larger audience, see the text boxes "OOP Made Visual: Digitalk's Look and Feel Kit" on page 112 and "Smalltalk About Windows" on page 114.*]

A Framework for the Future

One of the promising new concepts in object-oriented design—being actively explored today in Smalltalk as well as in other

OOP Made Visual: Digitalk's Look and Feel Kit

Ellen Ullman

Object-oriented programming is highly conceptual. Based on the ideas of encapsulation, inheritance, and polymorphism, OOP may seem to be the headiest and least visual of programming models. But the Look and Feel Kit from Digitalk manages a difficult task: It makes visible the concepts and mechanisms of OOP.

Based on the Smalltalk language, the Look and Feel Kit is more than a tool for creating GUIs. It provides a platform for creating complex object-oriented applications with a minimum of coding. It can also be used to integrate application components from a variety of sources, whether they're written in Smalltalk or another language, creating a consistent, object-oriented user interface. The screen at right shows an E-mail application in the process of being developed.

The development environment, running under Windows, has the familiar GUI. An iconic palette shows available categories of application components. The categories include windows, panes, buttons, menus, and tools. The palette is fully extensible and, when shipped, may include a category for database access, offering components for communication with Structured Query Language databases.

After you select a category, the palette shows the available components in that group. For example, selecting the panes category brings up options that include text, graphics, and list panes. The window category offers application windows and dialog boxes, and there is a range of buttons and menus. The tool category includes what are in essence fully functional "applets": a file accessor, an organizer tool for grouping collections of objects such as files, and the important dynamic link library (DLL) accessor for accessing modules written in other languages.

To create an application with the Look and Feel Kit, you drag selected components into the application editor window, placing them where you like. You can move components later. Pixel-level movement and alignment tools let you pinpoint placement.

After helping you arrange the graphical elements, most "visual" application tools stop being visual at that point. To get functionality behind components, you usually have to start writing code. But it's here that the Look and Feel Kit gets interesting.

To add functionality to a component, you first select an object. This elicits a pop-up menu showing the messages this object can send and receive. You select a message from the menu, and you get a "wire," which you then draw to the object that will receive the message. The wire is the communication line between objects. When you connect the wire to the receiving object, you get the receiving object's menu of messages. Selecting a message on the receiving object completes the connection. For example, a List Pane component showing the contents of a directory may send a `selected` message to a File Accessor com-

FOR MORE INFORMATION

Acumen Software
2140 Shattuck Ave.,
Suite 1008
Berkeley, CA 94704
(415) 649-0601
fax: (415) 649-0514
Circle 1114 on Inquiry Card.

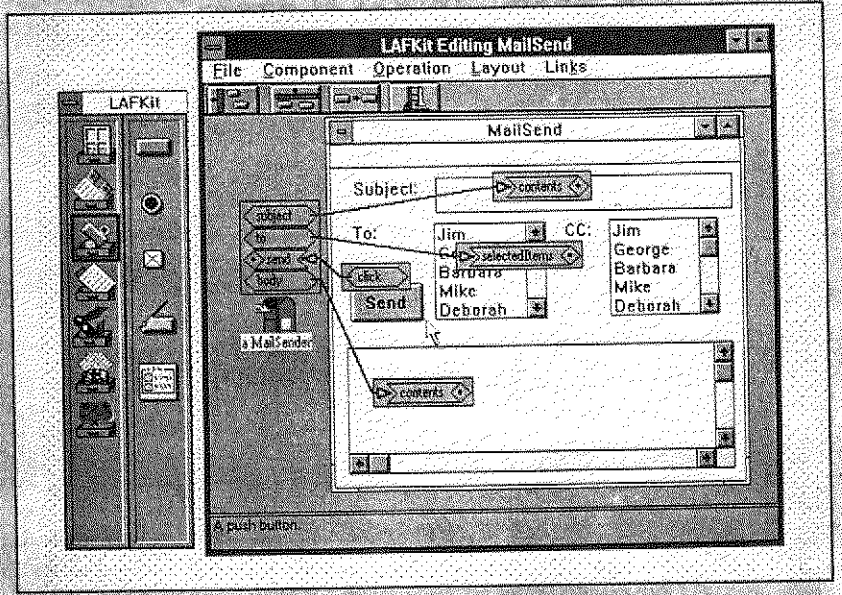
Digitalk, Inc.
9841 Airport Blvd.,
Suite 600
Los Angeles, CA 90045
(213) 645-1082
fax: (213) 645-1306
Circle 1115 on Inquiry Card.

ParcPlace Systems, Inc.
1550 Plymouth St.
Mountain View, CA 94043
(415) 691-6700
fax: (415) 691-6715
Circle 1116 on Inquiry Card.

languages and environments—is the concept of a framework. In an object-oriented environment that supports inheritance, reusable software that implements a single concept frequently takes the form of a specialization hierarchy in which the superclasses are more abstract (e.g., the Smalltalk classes `Collection` and `Number`), with certain operations deliberately left to implementation by more concrete subclasses (e.g., `Array` as a concrete subclass of a kind of `Collection`, and `Integer` as a kind of `Number`). These holes in the superclasses (called *virtual functions* in C++ terminology) are an important part of the design.

A framework is a generalization of this idea to a group of classes working together. For example, the Smalltalk model-view-controller framework consists of three abstract superclasses that provide little more than definitions of how the concrete subclasses should work together, plus some bookkeeping code and default implementations of the most common operations. You reuse a framework by writing new concrete subclasses and combining existing subclasses in new ways.

Another example of a framework involves the notion of a discrete event-driven simulation, in which objects interact to represent tasks, workers, locations (where tasks are carried out by



Digitalk's Look and Feel Kit lets you develop applications—like this E-mail program—by choosing components and connecting them with "wires."

ponent, which, in turn, accesses a file.

The process of drawing wires between components continues, object to object, until you have described all the lines of communication. The Look and Feel Kit displays some wires in red, indicating incomplete connections—those that require argument input. Complete connections are shown in green. You can test these connections by launching the application at any time.

When you are satisfied with what you have constructed, you save the application. You can then use it in three ways: You can add it to the palette, extending the available programming components; you can execute it as an .EXE file; or you can make the application a DLL for distribution.

The Look and Feel Kit, with its so-called wires, externalizes language-level objects. By grabbing a component, you immediately see its graphical elements. You also "see" its message capabilities—what are usually the conceptual, nongraphical aspects of an object. In a programming world where code is invoked through the passing of messages, the wires provide a literal, visual representation of the OOP procedural model.

By drawing connections, you can create complex applications with a minimum of coding. The determining factor in this development process is the quality and completeness of your component set. You can extend the set with applications you have created using the Look and Feel Kit, with Smalltalk objects you have written, with components purchased from third parties, and with DLLs. With a thorough set of components, you should be able to construct nontrivial applications, relying mainly on visual programming.

After the current Windows release of the Look and Feel Kit, Digitalk plans to ship an OS/2 version in October or November. This powerful, interesting development tool should add momentum to the OOP movement. And it may win some converts to the Smalltalk cause.

Ellen Ullman is a San Francisco–based associate news editor for BYTE. She can be contacted on BIX as "ullman."

the workers), and statistically based schedules for introducing tasks and workers. New components, specialized tasks, workers, and schedules can be described in order to reuse the general framework to create specific simulations. This concept is described fully in the book *Smalltalk-80: The Language* by Adele Goldberg and Dave Robson (Addison-Wesley, 1989).

The other Smalltalk idea receiving attention today is that building software is building systems. Software should have the same property as a fractal design: Assemblies built out of parts should have the same qualitative nature (such as definable inward and outward interfaces) as those parts. Developers must realize that they cannot predict all the ways that a piece of software will be used or all the ways that it will be ported to use the facilities of new environments.

Smalltalk in the Marketplace

One of the powerful ideas that has attracted new attention as a result of the development of object-oriented software technology is the notion of reusable, combinable applications. Today, this idea is promoted at three levels: (1) operating systems, such as Unix pipes and `fork/exec`; (2) window systems, by way of interapplication communications conventions (e.g., Apple's Interapplication Communications, Microsoft's Dynamic Data Exchange, and the X Window System's Inter-Client Communications Conventions Manual); and (3) independent software architectures (including low-level ones such as Microsoft's dynamic link libraries and Sun Microsystems' shareable libraries, as well as high-level ones such as Patriot Partners' Constellation project and ParcPlace's object model and frameworks approach).

Many believe that the discipline of defined, published interfaces—which the object-oriented approach naturally promotes—will create a new marketplace for reusable software components. However, from our experience with many developers and users of Smalltalk systems in many environments, we think the key economic shift will be in a different area.

A public market is a loosely organized environment. Components placed in a market will face a wide variety of demands, and even well-designed components with minimally constrained interfaces will have trouble attracting a critical mass of customers.

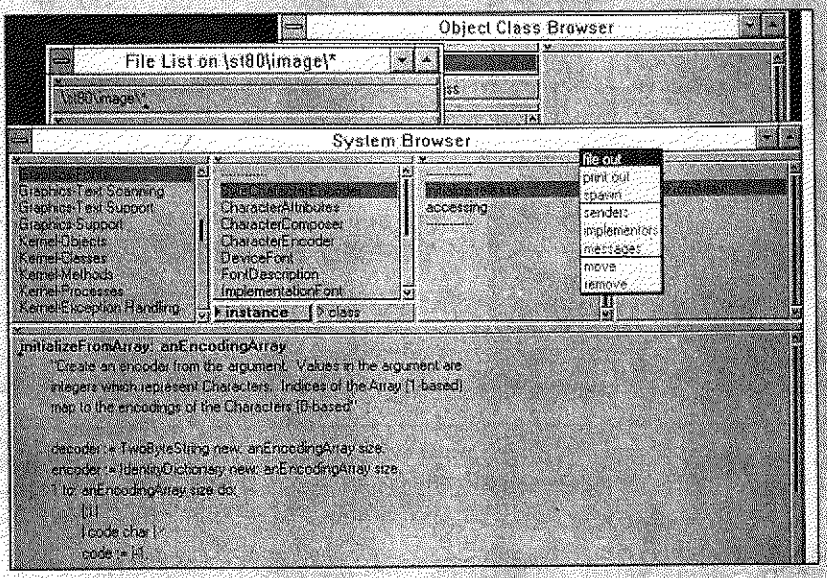
On the other hand, within a single organization, reusable components can be developed and redesigned to span a large fraction of their intended uses. In this way, the accumulation of

Smalltalk About Windows

Ben Smith

The Smalltalk environment has included windows since its inception. In fact, you might say that all the popular windowing environments grew out of the Smalltalk environment developed at Xerox Palo Alto Research Center (PARC). But, as with any evolving system, there are marked differences between the progenitor and its descendants.

Now, Smalltalk has recombined with the newest of the window environments, Microsoft Windows 3.0. The two major vendors of Smalltalk implementations for PCs have recently announced versions for Windows: Objectworks\Smalltalk for Windows from ParcPlace Systems, and Smalltalk/V Windows from Digitalk. While the core of both systems is Smalltalk, the Windows implementations are as different as the philosophies of the two companies.



Screen A: *Objectworks\Smalltalk for Windows looks similar to versions of the language that run on other platforms.*

A Question of Consistency

ParcPlace is the traditionalist; after all, the company *is* the tradition, since it spun off from the original group that developed Smalltalk at PARC. Objectworks\Smalltalk is a unique windowing environment with a mouse, window panes, scroll bars, and drop-down menus. You can use Objectworks\Smalltalk on a variety of platforms, and the window layout, icons, and window controls are always the same: the Objectworks style (see screen A).

Although this window style is not consistent with any of the newer and more widely used windowing systems, it has a great deal going for it. The most obvious feature is line wrap and rewrap: Long lines of text are wrapped around to the next line, breaking only between words. When you resize a window, the lines are rewrapped to reflect the new window size.

Another distinguishing feature of Objectworks\Smalltalk is its five-pane system browser window. (The system browser is the primary programming interface for Smalltalk.) Each pane is associated with a different function: class category, class, message category, message, and code editing. Each window pane has its own pop-up menu of operations. The pointers, icons, menus, and scroll bars maintain Objectworks' unique style on any platform.

Then there's Digitalk—the company that released the first commercial versions of Smalltalk (for DOS and then for Mac systems). Digitalk's Smalltalk/V Windows assumes that if you are programming for an established window environment, then you want to totally adopt that environment. In other words, if you develop a Smalltalk/V ap-

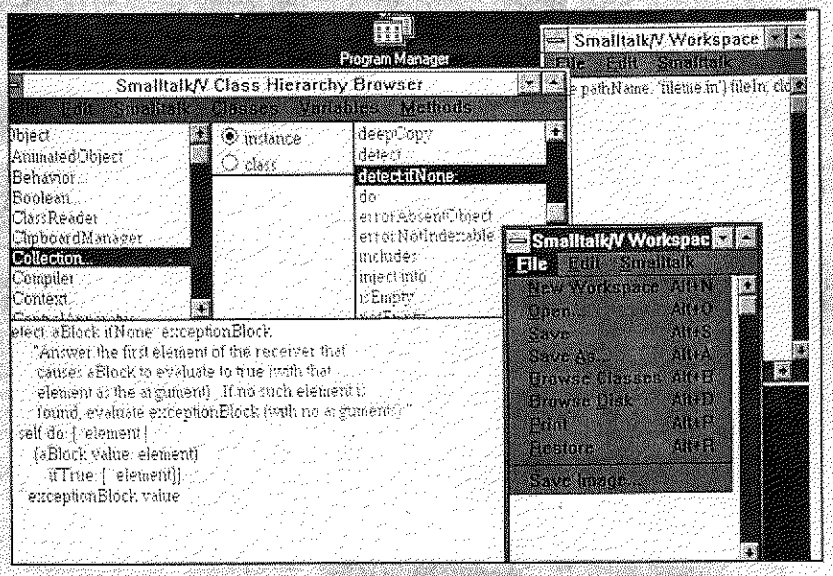
reusable code can become an important business asset and can be treated (appropriately) as an investment and a capital good, rather than simply as a cost (which is its present treatment).

In an object-oriented environment where inheritance is supported, it is not only individual components that are reused. As we have noted, the design of interfaces between objects is often more important than the implementation of functions within objects. Frameworks can capture the structural design of software objects that address a given (partial) problem domain. As such, the frameworks developed and reused within an organization will, over time, come to capture and eventually even define the expertise of the organization—and, as such, can contribute to the organization's ability to meet its customers' needs. (This is sometimes called *competitive advantage*, but it applies equally well in situations where competition is not involved.)

2001: A Smalltalk Odyssey

If we look into our murky crystal ball, how do we see software's use of object technology in the next decade? How do we see it evolving?

We hope that in 2001, objects will be boring. In comparison, radical ideas of past decades—that system software should be written in higher-level languages or in languages with strong type systems, and that computers can and should be seamlessly networked—are thoroughly accepted today. Whether to imple-



Screen B: *Smalltalk/V Windows applications are consistent with the Windows look and feel.*

plication for Windows, your application should look and act like a Windows application, not an application that merely runs inside of (and despite) Windows (see screen B).

The drawback to this attitude is that Smalltalk/V for the Macintosh looks and acts different from the versions for Windows, plain DOS, and the X Window System. The distinct advantage of Smalltalk/V for any environment is that you can take full advantage of that environment. Your applications will be consistent with the style guidelines for that environment. For example, Smalltalk/V Windows has full access to the facilities of the Windows application programming interface, including dynamic link libraries and Dynamic Data Exchange.

Tools and Classes

There's more to a Smalltalk implementation than a window environment and a language; there are the programming tools and the predefined class hierarchy. Smalltalk/V Windows provides fewer tools and a simpler class hierarchy than Objectworks\Smalltalk for Windows, but these limits are, in part, overcome by optional packages, like those from Digitalk (see the text box "OOP Made Visual: Digitalk's Look and Feel Kit" on page 112) and from third-party vendors such as Acumen Software. Acumen recently released a set of "user-interface construction kits" that let you develop interfaces for Smalltalk/V Mac, Smalltalk/V 286, and Smalltalk/V Windows programs—Widgets/V Mac, Widgets/V 286, and WindowBuilder/V, respectively.

Both Windows versions of Smalltalk maintain a text log of changes to the Smalltalk "image" (i.e., the Smalltalk gestalt of any moment). You can view the Smalltalk/V version of the log with the File utilities. With Objectworks\Smalltalk, you can view the change log as an object with a hierarchy that has separate instances for changes to classes, to methods, and to the system.

Both products provide a method for applying the changes of one project to another, a necessary operation if the system is to follow the objective of reusability. Both products also have an excellent debugger, as well as tools for file management, view management, and text management. As with all things, their styles differ: Objectworks maintains its own style, and Digitalk adopts the style of Windows.

Ben Smith is a technical editor for BYTE. He can be reached on BIX as "bensmith."

ment them is almost never an issue now, even though there is still plenty of discussion about how to implement them well.

In the same vein, we expect that 10 years from now, the object-oriented approach to software design and implementation will be an accepted, standard technique used in every language, library, database system, and operating system and will be taught in undergraduate computer science courses at every university. This is an issue of moving the technology further out into the world, and no major new thinking will be needed to accomplish it.

One significant technological advance will be that we will free ourselves even further from equating objects with the nouns in the problem domain. Some of the most remarkable advances in the usability of computer systems have come from recognizing that processes, as well as things, can and should be described, modeled, and manipulated. Therefore, we will see software objects being used to model time, places, actions, and events. We believe that this will lead to usability advances almost as dramatic as those resulting from the now-established window/icon/mouse/pull-down interfaces that were to a large extent inspired by the original Smalltalk work of the 1970s and 1980s. ■

L. Peter Deutsch is chief scientist and Adele Goldberg is president of ParcPlace Systems (Mountain View, CA). They can be reached on BIX c/o "editors."

MACAPP: AN APPLICATION FRAMEWORK

BY KURT J. SCHMUCKER

This application can significantly reduce Macintosh program development time

ONE FASCINATING and potentially far-reaching use of object-oriented programming is in the design of an application framework for a personal computer or workstation. Several examples of such frameworks exist, such as the Lisa Toolkit, discussed in "Software Frameworks" by Gregg Williams (December 1984 BYTE), and more are being designed all the time. This article examines one specific application framework for the Macintosh, MacApp—The Expandable Macintosh Application from Apple.

The average end user does not generally use or even know about application frameworks. They are tools for developers who design the software for end users. In theory, an application framework can be developed for any personal computer. However, they are especially useful on those with a well-defined user-interface specification.

WHAT IS MACAPP?

The MacApp framework is basically a complete, self-contained application that implements most of the Macintosh user-interface standard. It has menus that pull down and windows that scroll and can be moved about the screen, it works correctly with desk accessories and with Switcher, and it prints on the Imagewriter and the LaserWriter. The only things missing from a complete application are the contents of the windows and the items on the menus. An application framework is only the shell of a real application—a shell that you can easily customize into a true application. This customization process differentiates an application framework from a set of merely useful subroutines.

For example, let's examine the way in which an application framework supports undoing commands. MacApp knows that after you choose a menu command, the Undo command should reverse the effect of the command. But a general application framework can't know how to undo, or do, all the commands. These operations are accomplished with the dynamic binding present in an object-oriented language. The application framework "knows" about command objects and it knows that when a command is to be performed or undone, it should send the message DoIt or UndoIt to the current command object. The application framework defines the basic skeleton of the application, but it leaves the specifics—for example, the actual details of undoing the Double Space command—to the command object. To build a specific application from this framework, you need to design only the objects that perform these specific actions and then install them into the framework.

The framework knows in general what a Macintosh application is supposed to do. It knows how to make the menus work, how to give up control when a desk accessory is activated, how to scroll windows, and so on—all the things that are common to

(continued)

Kurt J. Schmucker, director of educational services for Productivity Products International (Severna Park Mall, H & R Block Office, 575 Richie Highway, Severna Park, MD 21146), teaches seminars on object-oriented programming. Kurt has written three books on computer science, including the forthcoming Object-oriented Programming for the Macintosh (*Hayden, 1986*).

Macintosh applications. The framework knows that the most recent command should be undone when you choose the Undo menu item and that the current selection should be highlighted when you activate a window. However, it doesn't know how to reverse the actions of particular commands or how to highlight the current selection. The *objects* you install in your customization of the application framework determine these actions. For example, to undo the last command, the application framework sends the message UndoIt to the current command object. The dynamic binding of this UndoIt message to a method at run time invokes the routine you have designed to handle undoing this particular command. The application framework proceeds without knowing what that command, or that selection, really is.

The application framework is more than just a skeleton with a fixed number of pluggable slots for commands and selection. Using the techniques of object-oriented programming, you can override every major decision (and many minor ones). Any application on this framework can take control at any decision point in the program by overriding the preprogrammed method to perform a user-written application-specific method.

To give it this flexibility, the application framework is set up as a group of classes, or class library, that you can use and specialize while developing a new application. If you want your application to behave in some unique, specific way, you can add some new objects into the framework to provide this behavior. If you don't want anything unusual, the application framework will handle the application correctly as is.

The Basic Structure of MacApp

The class library that is MacApp contains more than 30 different classes and over 450 methods. (Figure 1 shows the inheritance structure of these classes.) However, if you understand the operation of just three of these classes—TApplication, TDocument, and TView—and seven of their methods, you will be able to build your own application on top of the MacApp framework. The class TApplication takes care of things that are the responsibility of the application as a whole. This includes launching the application, setting up the menu bar, deciding which documents to display in the "Open Which Document?"

(continued)

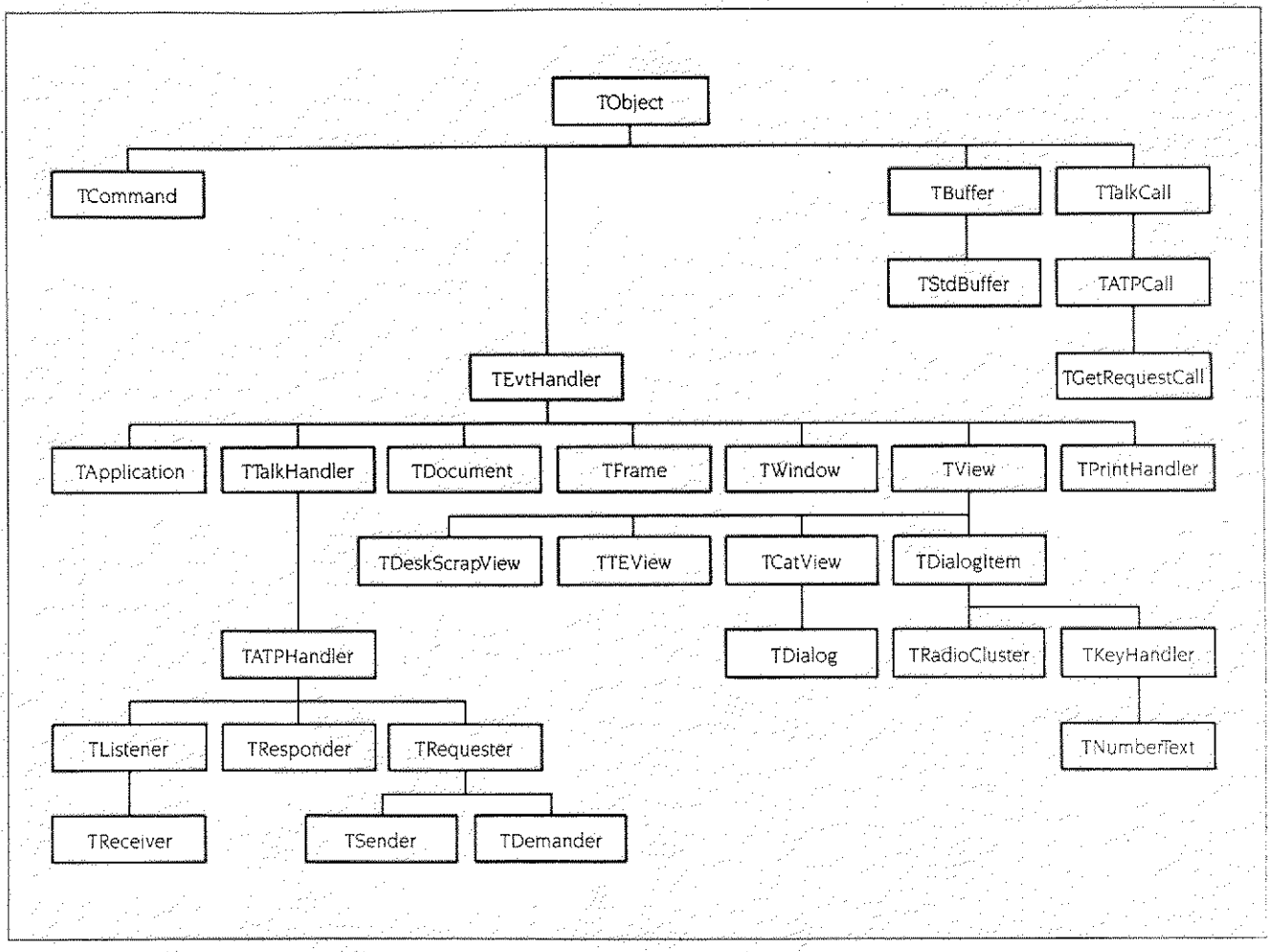


Figure 1: *The inheritance tree of the MacApp classes.*

dialog box, and so on. You design your own special subclass of TApplication, overriding whatever methods you choose in order to specialize any of these behaviors. One behavior you must always override is the type of document that holds your application's data (the method DoMakeDocument).

The class TDocument processes commands like Save and Close, which are specific to each of the documents that are open at any one instant. (MacApp applications can usually deal with multiple documents being open at once.) Two behaviors that you must override in your subclasses of TDocument are the types of windows that display the data stored in the document (the method DoMakeWindows) and the contents of the windows (the method DoMakeViews). (The DoMake-something MacApp methods are the ones you *must* override.)

The class TView takes care of everything inside your windows—drawing the images, highlighting the selection, handling mouse interaction with those images, and other things. TView knows when a portion of the window needs to be redrawn and when the selection should be highlighted. It doesn't know exactly how to do these things. It relies on you to override the methods that supply these behaviors in your subclasses of TView. These methods are Draw, HighlightSelection, and DoMouseCommand.

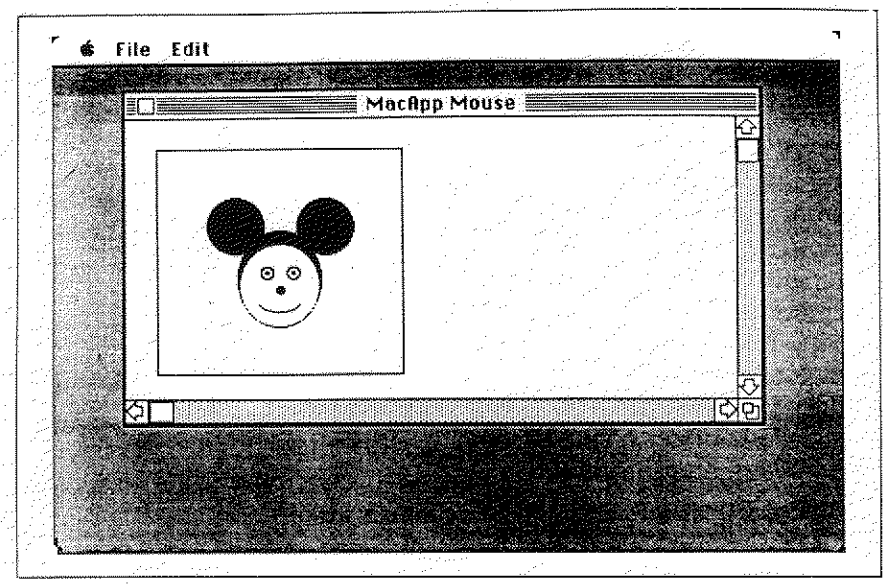


Figure 2: *SmallApplication—the smallest MacApp application.*

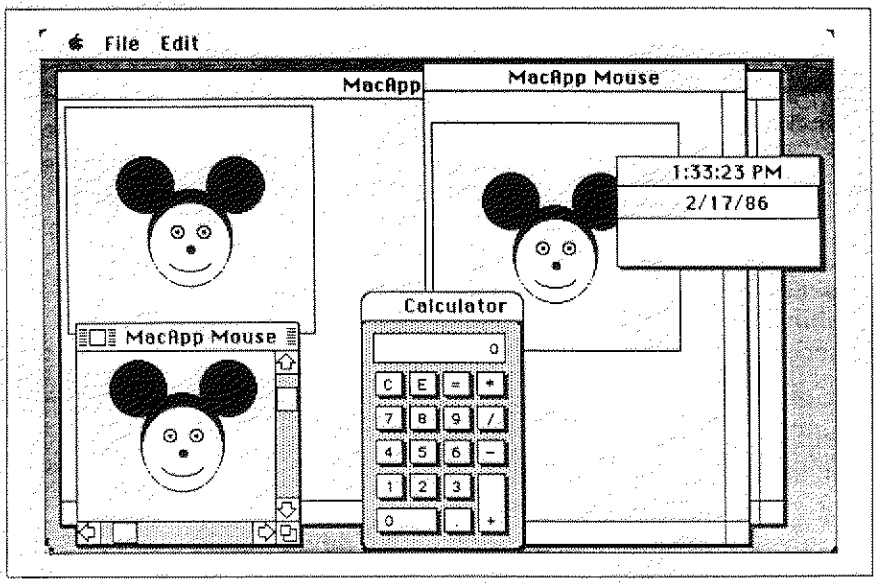


Figure 3: *MacApp applications typically work with multiple documents and always work correctly with desk accessories, even multiple ones.*

Developing an Application

To develop a MacApp application, you must design your own subclasses of TApplication, TDocument, and TView. It is traditional in MacApp programming to name these new subclasses so that you can easily determine their respective superclasses. Therefore, I have used the names TSmallApplication (a subclass of TApplication), TSmallDocument (a subclass of TDocument), and TSmallView (a subclass of TView). The application is called SmallApplication, and its entire source code requires only 87 lines of Object Pascal. (For a discussion of Object Pascal and other object-oriented languages, see my article "Object-oriented Languages for the Macintosh" on page 177.) Two printouts of screen shots from SmallApplication are shown in figures 2 and 3. [*Editor's note: The entire source listing for SmallApplication is available in a variety of formats. See page* 405 *for details.*] Let's look at two representative methods from this application—the DoMakeViews method of the class TSmallDocument and the Draw method of TSmallView.

DoMakeViews is one of the methods MacApp needs to access one of the classes designed specifically for SmallApplication. I call this kind of method a MacApp *hook* method. Listing 1 contains the full text of SmallApplication's DoMakeViews method. This method generates, initializes, and installs one instance of TSmallView. MacApp sends the message DoMakeViews precisely so it can obtain one of these and use it to draw inside the window. If this method seems rather short, that is a common characteristic of object-oriented programs, especially those

Listing 1: *The full text of* DoMakeViews.

```
PROCEDURE TSmallDocument.DoMakeViews(forPrinting: BOOLEAN); OVERRIDE;
VAR smallView: TSmallView;
BEGIN
   NEW(smallView);                  { Create a new instance of TSmallView }
   smallView.ISmallView(SELF);      { Send new view object its init message }
   SELF.fSmallView := smallView; { Install this view object in document }
END;
```

Listing 2: A *procedure that overrides* TSmallView's Draw *method to draw a picture of a mouse.*

```
PROCEDURE TSmallView.Draw(area: Rect); OVERRIDE;

   FUNCTION MakeRect(top, left, bottom, right: INTEGER): Rect;
   VAR r: Rect;
   BEGIN
      SetRect(r, left, top, right, bottom);
      MakeRect := r;
   END;

BEGIN
   PenNormal;
   PaintOval(MakeRect(74, 72, 139, 127));       { Outline of the mouse head }
   EraseOval(MakeRect(84, 74, 138, 125));       { Outline of the mouse face }
   FrameOval(MakeRect(109, 84, 129, 115));      { Mouse mouth (part 1 of 2) }
   EraseRect(MakeRect(109, 84, 123, 115));      { Mouse mouth (part 2 of 2) }
   FrameOval(MakeRect(98, 87, 107, 96));        { Left eye }
   FrameOval(MakeRect(98, 104, 107, 113));      { Right eye }
   PaintOval(MakeRect(101, 90, 104, 93));       { Left pupil }
   PaintOval(MakeRect(101, 107, 104, 110));     { Right pupil }
   PaintOval(MakeRect(111, 97, 117, 103));      { Nose }
   PaintOval(MakeRect(53, 52, 91, 90));         { Left ear }
   PaintOval(MakeRect(53, 110, 91, 148));       { Right ear }

   FrameRect(MakeRect(20, 20, 170, 180));       { A bounding rectangle }

END;
```

designed to be overridden for many different purposes. Instead of hard coding many decisions, the designer of a class will make each such decision a method. You can change such a decision by creating subclasses and overriding the appropriate method.

The Draw method of the TSmallView class is a method for which MacApp cannot possibly provide a generic version. You can't draw anything in a window that would be useful to all Macintosh applications. In such cases, MacApp provides a stub method that does nothing, a *null* method. You don't have to override a null method like you do a hook method, but if you don't override this one, part of your application may appear to do nothing. The code in listing 2 overrides TSmallView's Draw method to draw a picture of a mouse.

If you continue this process for five other methods, you will have developed SmallApplication, an application that draws a picture of a mouse. SmallApplication is a stand-alone Mac application that works correctly on 128K-byte and 512K-byte Macs, the new Mac Plus, and the Mac XL. It works with Switcher and with any number of desk accessories, prints on the Imagewriter and the LaserWriter, supports multiple documents, and allows you to resize and move windows and use menus. As trivial as the application itself may seem, it does illustrate the flexibility of the MacApp framework.

The Benefits and Costs of Using MacApp

Early studies indicate that MacApp can reduce application development time by a factor of four or five. MacApp also decreases the amount of source code you need, again by a factor of four or five. It maintains consistency with respect to the Macintosh user-interface standard and provides error handling and an interactive debugging facility, which are useful during development. It provides a conceptual framework that lets you concentrate on your application rather than on Macintosh internals.

Some feel that these gains are at the expense of performance in the finished application and of a large amount of additional memory. In fact, many MacApp programs actually run faster than their non-MacApp versions, despite the run-time overhead of messaging. MacApp applications are usually somewhat larger than their non-MacApp versions—about 10K to 15K bytes. But for most end-user applications, this is not a large penalty when weighed against the decrease in development time. ■

PROGRAMMING EXPERIENCES

BY LARRY TESLER

Programmers using object-oriented languages say the benefits make the learning worthwhile

WHAT IS IT LIKE to write a program in an object-oriented language? I posed that question to several people who program in Objective-C, C++, Object Pascal, and Smalltalk, hoping to gain insight into how different programmers think about object-oriented design. Their experiences had more in common than you might expect.

I asked each person to describe his project and discuss how object-oriented programming affected its progress. Their recollections tended to support oft-heard claims that object-oriented languages can be a boon to large programming projects. The software development benefits stem from three properties of object-oriented programs: object-based modular structure, data abstraction, and the ability to share code through inheritance.

The term *modularity* refers to the factoring of a large program into units that can be modified independently. In an object-oriented system, every module is an object, that is, a data structure that contains the procedures that operate upon it. Object-oriented design is the process of identifying objects that constitute a useful model of the problem at hand. In the early stages of designing a program, the need to partition the problem into objects stimulates the designers to identify its principal constituents and to specify their behavior and interaction.

Data abstraction is the process of hiding a data structure behind a set of procedures through which access to the data is forced. In this way, the "concrete" representation chosen by the programmer is replaced by an "abstract" catalog of available operations. The advantage of data abstraction is that at any time the programmer can change representations without having to change other programs that relate to the operations. Data abstraction is a natural concomitant of object-oriented programming because each object contains not only its data structure but also the procedures that operate upon it. These procedures, often called methods, are usually the only aspects of the object accessible to other objects.

All object-oriented languages can share code through *inheritance*: that is, object-oriented languages provide the ability to define one type of object as a variation of an existing type. The new object type is called a *subclass* of the old, and the old type a *superclass* of the new type. Objects in the subclass inherit all the properties of the superclass, including the implementations of methods. The subclass can define additional methods and redefine old methods by providing so-called *overrides*. By using inheritance during the development of an object-oriented program, code can be shared among similar objects. Later, certain kinds of enhancements can be made simply by creating new object types as variations of existing ones.

A WINDOWING SYSTEM

The first person I interviewed was Gary Walker, Manager of Primary Interaction Development in the Distributed Systems Group at Burroughs Corporation in Boulder, Colorado. He and his group of nine programmers were assigned the task of implement-

(continued)

Larry Tesler, currently Manager of Advanced Development at Apple Computer, previously managed the development of Lisa applications, the Lisa Toolkit, and MacApp. He can be contacted at Apple Computer, 20525 Mariani Ave., Dept. 5770, Cupertino, CA 95014.

ing a general windowing environment, featuring menus, check boxes, buttons, and the other trappings of a see-and-point user interface. After conducting a comparative study of the available object-oriented languages, his group chose C++, an object-oriented extension of C inspired by Simula-67 and developed by Bjarne Stroustrup at Bell Laboratories in Murray Hill, New Jersey. Only object-oriented languages were considered for the project. "In a windowing system," Walker explained, "you want to instantiate objects for windows, each with its own private data. By defining separate types of windows as different classes, they can inherit common characteristics and still possess their own special properties."

Walker found data abstraction to be the most significant advantage of using C++. Smalltalk and some other object-oriented languages force data abstraction upon the programmer by hiding the internal structure of one object from other objects. For example, to move a chess piece, a Smalltalk program must invoke a method such as move_to, passing the destination square as a parameter. It cannot use an assignment statement to modify the data structure describing the chess piece's position. The advantage of the restriction is that both the representation of chess pieces and the implementation of move_to can be changed without having to alter the code in other objects that access them.

Unlike Smalltalk, Object Pascal and C++ allow objects to access part or all of the internal data of other objects. However, many textbooks warn against direct data access except when performance considerations are paramount. Walker's group found through experience with C++ that interobject direct data access is usually a detriment to modularity. "If you want to get at somebody else's variables," he said, "you should go through access functions [methods]."

Another property of object-oriented programs that benefited the windowing system project is modular structure. It gave the designers the ability to create what Walker calls "isolated

(continued)

worlds of data and functions."

Walker also suggested a more pragmatic advantage of modularity based on objects: cutting down on the number of global variables in the program. The advantage of avoiding global variables in an interactive system is that multiple instances of each object can easily be created. It would be quite difficult to support multiple windows if the data describing a window resided in global variables. According to Walker, if you follow the advice of many software engineering books and avoid global variables, you usually end up passing too many parameters to functions. With C++, Walker explained, data can be "private" to an object, and all functions of that object can access its data without passing parameters.

Having heard Walker mention inheritance as a key factor in his choice of the object-oriented paradigm, I asked him for an example of its use in the windowing system. He cited the class Menu, a data abstraction with several subclasses, including VerticalMenu, RadioButtons, and CheckBoxes. The system displays each type of menu a different way, and the user interacts with each a bit differently. But all serve the same basic purpose: They give the user choices, and they report the user's choice to the object in the application program.

Some methods of Menu are inherited by the subclasses without modification, while others are overridden by special implementations in each subclass. An example of an inherited method is selectionTitle, which returns the string containing the user's menu choice. The implementation of selectionTitle is shared by Menu and all its subclasses. An example of an overridden method is prompt, a function whose arguments are the text strings that represent the choices available in the menu. For example, my_menu.prompt("sherbert","cheese cake","torte") specifies the choices in a dessert menu. Each subclass of Menu implements its own version of prompt. The version in the class VerticalMenu displays a list of the strings in a style similar to Macintosh pull-down menus, while the version in the class CheckBoxes displays the strings side by side with a check box beside each one, similar to Macintosh dialogs.

The variable my_menu is declared to be of the type Menu, but at different times during execution its value may refer to objects of different subclasses of Menu. Whenever the my_menu.prompt is executed, it will invoke the version of prompt associated with the class of the object that is currently referred to by my_menu. This is one of several cases where Walker's group found a use for the so-called polymorphic property of objects. Polymorphism refers to the ability of one procedure call to invoke different procedures at run time depending on the type of one of its parameters. In object-oriented languages, polymorphism is achieved by letting different classes implement methods that have the same name and formal parameters but different implementations.

The ability of subclasses to inherit from superclasses can also simplify the maintenance of large object-oriented programs. The Burroughs team found that by making a change to the superclass, in effect they changed all the subclasses at once, and if they made changes to one of the subclasses to get distinctions they wanted, the code in the superclass and the other subclasses remained safe.

Walker's group was not alone in that finding. I heard similar claims from Seth Snyder and Dale Peterson of Recording Studio Equipment Company based in Miami, Florida, who used an object-oriented language to implement an integrated application that controls a spectrum analyzer while managing time billing for a recording studio. According to Snyder and Peterson, when new features had to be added to their program, they were able to add them reliably, without any risk of affecting the performance of features they had implemented earlier.

A Shipboard Navigation System

Carl Nelson, a computer consultant in Seattle, Washington, was approached

(continued)

by a group of investors for his assistance in building a computer-assisted navigation system. The envisioned system, to be installed on boats in coastal waters, would consist of a Macintosh connected to a loran. A loran collects data on a ship's position from a radio receiver tuned to three or more land-based transmitters. Using a combination of triangulation and dead reckoning, it displays the ship's position and bearing on a simple (one- to three-line) display. The captain can key in the latitude and longitude of points along the desired course, and thereafter the loran will display the current heading and the distance to the next point in the course. If connected to an autopilot, the loran can command it to steer the vessel along the planned route.

The clients told Nelson that even though the loran and autopilot are mainstays of navigation for many boat owners, the equipment can be tedious and time-consuming to use. The digital information on the display does not relate to a position on a navigational chart at first glance. A "what you see is where you are" system—one that displays the chart on the screen with the present course lines superimposed on the image—was needed. Such a system would allow a navigator to plan a course on the chart with a mouse and then would transmit the coordinates electronically to the loran. The system would save time, increase accuracy, and avoid problems that arise when incorrect coordinates are entered.

The entrepreneurs used a Thunderscan digitizer to transfer images of nautical charts into MacPaint files, and they wrote a utility program to convert those files to a format usable by the application. One of the investors already had a Macintosh connected to the loran on his boat and recorded the telemetry of one day's voyage on a floppy disk. That disk enabled Nelson to test his program in the comfort of his office. For testing, Nelson used two computers. The main computer displayed the chart and allowed the course to be specified with a mouse. The other Macintosh served as a loran simulator, playing back the recorded telemetry through one of its serial ports to the main computer.

All that was left was to program the application and the simulator. Because he had only four months from project start to public demonstration, Nelson needed a software development environment that enabled rapid prototype development and implementation. He chose MacApp, an object-oriented software framework for the Macintosh (see "MacApp: An Application Framework" on page 189), and Object Pascal, the only language available then (mid-1985) that could be used with MacApp.

To understand MacApp, you must be familiar with certain standard concepts underlying Macintosh applications, including the concepts of document, view, window, and command. A document in the Macintosh corresponds roughly to a file in a traditional computer. The programmer must design a file format for storing it on disk and a data structure for storing it in memory. The programmer must also provide one or more ways to represent the document visually on the display and on the printed page.

Each different visual representation is called a view. For example, an array of floating-point numbers can be viewed as a tabular column of text containing digits and decimal points, or as a pie chart with shaded wedges of varying size. The size of a view often exceeds the size of the screen, but you can see portions of it through a window that you can scroll and resize. Using the mouse and the keyboard, you can issue commands that change the document. The changes are reflected in all views of that document that are presently displayed.

MacApp defines the abstract classes Document, View, Window, and Command, corresponding to the above concepts. A class includes a set of methods that define what the class can do. For example, a document can open and save, a view can draw and print, a window can resize and move, and a command can do and undo. To use MacApp, you must structure the application in a modular fashion in terms of these objects. Once that is done, the application can inherit an extensive library of user-interface and

(continued)

error-handling facilities.

According to Nelson, the framework provided by MacApp gave him a structure to plug things into. As he studied the navigation problem, he asked himself, "What do I have in this application that maps onto objects supplied by MacApp?" After identifying all the concepts that mapped easily into MacApp objects, he found that the whole user interface was accounted for. The only code that remained to be designed was that which manipulated internal data structures unrelated to the user interface.

In the navigation application, the most important subclass of the class View was easy to identify: a digitized chart with latitude and longitude lines. The window in which that view was displayed was a little harder to design, because it had to provide nonstandard controls for scrolling around a spherical world. The command objects were easily determined by enumerating the commands available in the user interface, such as place marker and show navigation info. The choice of document objects was not so clear-cut.

A document in MacApp is an object that manages the principal data structures of an application both in RAM and in file storage. In Nelson's application, several different files are employed, including the digitized nautical chart image with added annotations, and a trip file, which consists mainly of the trail of coordinates recorded during a specific voyage. Nelson had to decide whether the document object of his application should be of the class NauticalChart or of the class Trip, or whether his application should support both kinds of documents. He based his decision on an analysis of the operations associated with each type of object. For example, he wanted the client to be able to save the history of a trip in a file and then reopen that file by clicking an icon in the Macintosh Finder. But he also wanted the client to be able to open a chart file to review the annotations that had been made on the chart. He concluded that both the trip and the chart are appropriate document objects, and his application defines both as subclasses of Document.

The chart file consisted of a digitized image plus markers indicating significant locations such as reefs and buoys. Once the program was running, Nelson and his client realized that not all markers should be associated with the chart file. It made sense for a marker labeled "lighthouse" to be stored with the chart, but a marker labeled "caught 30 lb salmon" really belonged with the trip. Nelson decided to divide all markers into two subclasses of the object class Marker, namely, TripMarker and ChartMarker. He analyzed what the two kinds of markers had in common—for example, the display algorithm and the routines to edit an annotation—and implemented that common behavior in the superclass Marker, from which the subclasses could inherit it. He also determined what differentiated them—for example, the shape of the displayed icon and the file used for storage—and implemented that special behavior as overrides in the subclasses. Nelson called the differentiation process "pushing down the details" from superclass to subclass (see figure 1).

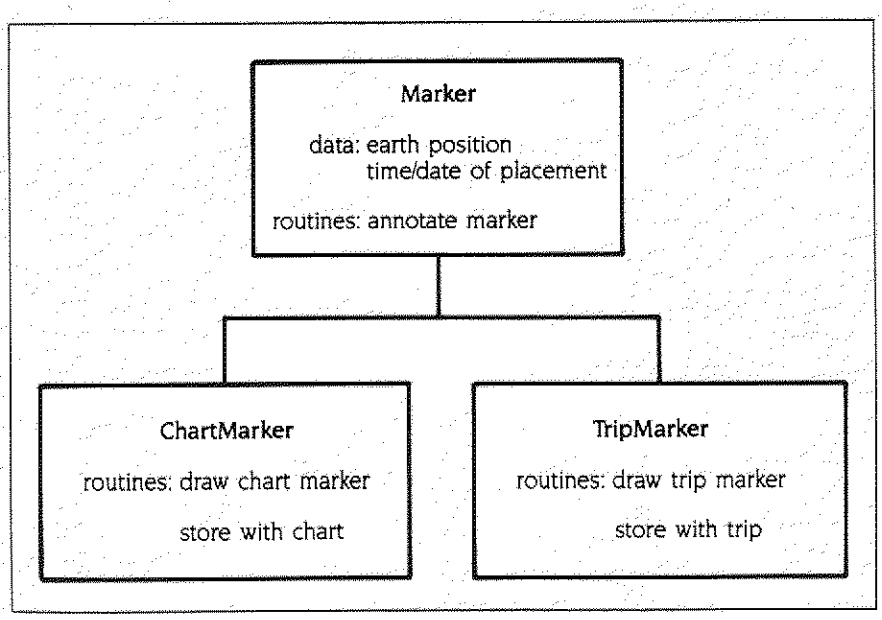


Figure 1: *An example of class hierarchy.*

A CAD System

At Artecon Inc. in Carlsbad, California, a group of 20 programmers led by Dana Kammersgard used an object-oriented language on Sun-2 and Sun-3 workstations in the development of ArteMate, an integrated CAD and office automation system. To make the system as portable as possible, Kammersgard's graphics group coded their routines according to an industry standard called GKS (Graphical Kernel System). GKS provides a way to construct images by transforming and combining primitive forms such as lines, polygons, curves, and ellipsoids. The standard specifies a device-independent set of procedure calls, leaving to each implementation the task of interpreting those calls in a manner appropriate to the available output devices.

According to Kammersgard, his team wanted the CAD portion of ArteMate to display two-dimensional and three-dimensional graphics on a wide variety of plotters and screens. To obtain that flexibility, an object-oriented approach seemed best. The language they chose for their implementation was Objective-C, developed by Productivity Products International of Sandy Hook, Connecticut, and available on a variety of computers and operating systems.

The first question Kammersgard's group addressed was how to organize the code for a number of graphics devices, including the CalComp 1043

(continued)

and 1044 and the HP 758X models, in such a way that it could perform both input and output to a number of black-and-white and color display systems, including the Sun Color Graphics Processor and the IBM 5080. The programmers decided that each type of device should be represented by a different type of object. Accordingly, they defined Objective-C classes such as SunGP and CalComp1044.

At different times during program execution, a program variable can contain pointers to different device classes. For example, if dev refers to an instance of the class SunGP, the statement dev poly__line: coordList invokes a device-specific method in class SunGP to display a polygon on the Sun screen. If dev is later assigned a reference to a CalComp 1044, the statement dev poly__line: coordList invokes a device-specific drawing method in the class CalComp1044 to drive the pen plotter along a polygonal path. To support a new device, the the programming team can simply define a new class without modifying existing code.

Kammersgard says that where they could take advantage of special hardware features, they implemented a device-specific method in the class. For example, the method poly__line normally has to apply transformations to the coordinates supplied in its parameter list to account for the visual perspective of the viewer. To calculate these transformations involves matrix multiplications, which are time-consuming operations in a conventional computer. Because the Sun Graphics Processor implements a three-dimensional transformation pipeline in hardware, the class SunGP overrides the standard implementation of poly__line, substituting a version that is shorter and faster than transformations performed wholly in software.

Like biologists who classify life forms into species, group similar species into a genus, group related genera into a class, and so on, object-oriented programmers design hierarchies of classes according to the similarities and differences they perceive between objects. In the Artecon system, specific output devices are the species of the graphics kingdom, and company product lines are the genera. Since different devices from the same manufacturer often have similar interface specifications, Kammersgard's team defined the class CalCompPlotter as a superclass of both CalComp1043 and CalComp1044. They moved methods common to both models up to the superclass and left model-specific methods in the subclasses. In a similar fashion they added generic classes like HPPlotter, SunDisplay, and IBM50SeriesDisplay to the class hierarchy. By sharing as much code as possible between device classes, they were able to reduce program size and development time considerably.

The hierarchy of device classes continues for two more levels. At the level above product lines, all kinds of plotters are grouped into one class, and all kinds of interactive displays into another; display classes implement methods for user input, while plotter classes do not. At the highest level is the class GKSWorkstation, which is the ancestor of all other device classes. At that level, device-independent operations are implemented—for example, the GKS primitives that change display attributes in data structures in memory without communicating to the devices.

In any graphics application, another obvious application of objects is to represent the graphical components of the drawing. For example, all ellipsoids ought to be instances of the class Ellipsoid, and all cylinders ought to be instances of the class Cylinder. In the Artecon system, all geometric modeling classes are grouped together under a superclass called GeometricObject. Geometric objects respond to messages such as draw, rotate, and store.

But a CAD system must do more than a simple drawing program. It must allow the user to indicate relationships among design components. Kammersgard's group found themselves adding "links" to geometric objects and to other objects within the system, such as instances of the class ViewPort. After a while, they realized

(continued)

that the various implementations of links could be combined by embodying Geometric_Object and ViewPort in a new superclass called AssociativityObject. An associativity object contains a set of links and supports operations such as add_link, remove_link, and modify_link. A member of any subclass, say, Cylinder, inherits the ability to contain links as well as the routines for manipulating them. Adding the class AssociativityObject required a modest restructuring of existing code. According to Kammersgard, it is common to restructure the class hierarchy to take advantage of newly discovered opportunities for sharing code through inheritance.

A Knowledge-Based Application

Bill Hutchison, a behavioral psychologist living in Silver Spring, Maryland, is implementing a knowledge-based system on the IBM PC. The system organizes information in a way that allows a seemingly rational response to stimuli. After considering a number of development systems, Hutchison decided upon Methods, a Smalltalk dialect developed by Digitalk Inc. of Los Angeles, California. I spoke to Hutchison after he had been using Methods for four months. "I like the way I can think about the problem," he said. "I map out the general problem in my head and can almost extract the objects from how I write it down in English. I make an object for each physical thing, process, or activity that I am dealing with."

I asked him if Smalltalk was difficult to learn. Hutchison, who has programmed extensively in assembly language, COBOL, BASIC, PILOT, and PLANIT, said he found Smalltalk "the most natural way" to program. He admits, however, that most of his learning time went to mastering Smalltalk's extensive class library. Large libraries are typical in object-oriented systems because they are extremely easy to build and maintain using subclassing and inheritance. The library that comes with Methods includes classes that are similar in purpose to those of MacApp. That allowed him to implement the user interface of his application easily and give it fancier features than he had first thought possible.

Hutchison said he structured the application's objects in a modular way. Knowledge is stored in association networks that relate situations, conclusions, and responses. He first developed a basic Network class able to represent simple domains, and he said that doing so was not as difficult as he had expected. Later, when he decided to tackle more difficult problems, complex networks became subclasses of the basic version. The first subclass he defined was InteractionNetwork, which adds the ability for parts of networks to interact with each other. That class was itself subclassed to define MultiResponseInteractionNetwork, which permits the system to respond along multiple dimensions.

At each stage he had to restructure existing definitions a little to allow the new class to inherit as much as possible from the old classes. The modular structure of the application made it easier to change one part without affecting others. "Sometimes," Hutchison said, "a radical change that I was dreading took me only an hour or less to accomplish." But Hutchison added that to make the program that modular, he had to develop the discipline to confine knowledge of an object's internal structure to its own class—only after having done that could he make changes to an object's structure without affecting others.

Object-Oriented Future

Certainly, object-oriented programming offers a great deal to software developers who want to manage large software projects or create prototypes quickly. Now that several suitable languages are widely available, many programmers will likely invest the time necessary to acquire the skill of using them. The interviews I conducted encouraged me to believe that these languages can be applied effectively in diverse situations by people of varied technical backgrounds. Even though the learning curve is high, most programmers can easily exploit the full potential of object modularity, data abstraction, and inheritance offered by object-oriented languages. ■

It doesn't make sense to measure a group's productivity by the output of 10% of its time. There must be a better way.

MANAGING SOFTWARE DEVELOPMENT

by Patrick Brown

The real difficulty in managing software development is the fuzzy concept of exactly what is being managed. Development is typically managed using a phased development model, and a common form of productivity measure is the KLOC/month. Yet it is not clear that either phase development models or productivity measures actually represent development activities.

Development is often broken into several discrete efforts, sometimes called phases. Common phases are requirements definition, external design, internal design, code and unit test, and integration and system test. Managing this process involves the formal review of the products of each phase.

Requirements, external design, and internal design produce documents describing the results of analysis and design activities, constraints, assumptions, and future work plans.

The product of code and unit test is the delivery of tested code to the integration/system test group. Other than a check to make sure all components have been delivered, this phase is normally not reviewed.

Integration and system testing usually involves exercising all or most of the delivered system and either formal or informal evaluation of the test results. Generally, a sign-off or formal acceptance of the product ends the testing period.

It is interesting to note that a number of diverse skills are required to use the phase development process. Analysis and design are intended to be the primary activities in the requirements and design phases. The product being reviewed, however, is always a result of documentation skills. Documentation skills are necessary since poor documentation can ruin a good design. In the code and unit test phase the products are indeed program components that require coding and testing skills. The integration test phase has no tangible product but requires experienced testing skills and good data processing background.

There are severe problems with the phase management process. Documentation of the product is becoming an enormous burden. It is not uncommon for projects to produce design documents thousands of pages long. Producing such documents costs far more than any value they may have to anybody, including the authors. Few development groups use external design documents after internal design efforts have been completed.

Internal design documents share the same fate. The cost of maintaining a current version of the document is simply too large to make it feasible. Most groups resort to informal methods—or worse, no method at all—to keep track of changed designs. The document is normally of less value to outside groups than to the authors. Usually, an outside group wants to know about a small part of the entire product. Should a person read a 1,200-page design document to find the format of a single interface?

Another significant problem with the process is that documents are rarely compatible with one another. In many cases the production of the external design document is a separate effort from the requirements document. Similarly, the internal design document is a separate effort from external design. This makes continuity in design difficult. It is a rare development group that can trace every feature in a product back to a requirements statement, or that can demonstrate that every requirements statement has been satisfied by some design component.

An approach to explaining the problems is that while the phase process demands documentation, the development process requires information. Documentation is not a substitute for information. This will be soundly endorsed by anyone who has waded through a few hundred pages of documentation looking for a parameter format.

MANAGING DESIGN PROCESS

Additionally, emphasis on documentation encourages managing the design process as if it were a publishing business. Instead of creating hierarchic levels of design detail, we are faced with the problem of publishing design documents—volumes 1, 2, and 3—with associated draft and publishing deadlines. An examination of work plans for design groups will uncover tasks like "produce preliminary draft" and "edit final draft." More appropriate tasks might be "integrate all design components" and "cross-check components for consistency and completeness."

Measurements of programming efforts are imprecise at best and in many cases probably meaningless. Though the industry recognizes a need for productivity and quality measurements, there are no clear ideas about what should be measured. A commonly accepted concept of productivity is lines of code per unit time. But using lines of code (or KLOC per month) creates a major problem in terms of consistency in counting. A few of the variables are different languages, comments, statements vs. card images, executable statements vs. all statements, and differing levels of complexity.

Even more important than inconsistency is the question of whether lines of code is an appropriate measure of software development. Coding, the production of lines of code, normally occupies about 10% of project time for new development and perhaps 1% to 3% of maintenance time. How can it make sense to measure a group's productivity by the output of less than 10% of its time? Certainly the design phase must have been productive or the project would not have been undertaken. A true measure of productivity must account for the activities of all phases.

It is not uncommon for projects to produce design documents thousands of pages long.

It may be argued that lines of code are used as a measurement because it is the only tangible feature of the delivered product. This argument, however, only reinforces the idea that the phase process is an inappropriate technique for managing software development. Each phase of the development should produce some tangible, measurable product.

Quality measurements suffer from similar faults. Typical quality measurements refer to error density and mean time to failure. These kinds of measurements are suitable for manufacturing environments, but software is not mass-produced. Error density in software might be analogous to measuring the number of defective bricks or the number of sticky doors in a new office building. The owner of a building is not interested in counting defective bricks, he is interested in how well the new building meets his business needs. Software quality is not measured by counting defective building blocks; it is a measure of how well the new system meets the needs of the business it was designed to support.

As sad as the current situation is, the future promises even more problems. Current techniques simply will not support emerging methods of development that bypass many traditional phases.

Take, for example, prototyping. Prototyping has received increasing attention in recent literature. The theory is that by quickly building a model of the target system, much of the requirements and design phase can be bypassed. If a prototype can be built in six weeks, it makes little sense to spend six to eight weeks producing a requirements document for the system. Spending additional time developing comprehensive external and internal design documents about an existing system may not make sense either.

AS CHEAP TO BUY AS DEVELOP

In addition, many companies are purchasing software instead of developing it. Not only is it nearly as cheap to buy code as to develop new code, purchase does not require the acquisition and maintenance of development resources. To what extent should a development organization provide design information to the software vendor? Internal design information may not be appropriate. External interfaces are certainly important for custom-built systems. The traditional phase development process does not easily deal with such questions.

Automatic application generators are advertised as able to increase productivity by large factors. But many generators produce neither usable internal designs nor source code. It is not reasonable to measure the output of an application generator in the same fashion as we measure a programmer writing applications. Once again, the traditional management process and traditional measurements do not easily handle this situation.

It would be nice if one management technique could handle all kinds of development work. Unfortunately, this is probably not feasible, but a few simple guidelines provide at least a consistent approach to the problems of diverse development methods.

- Manage what you produce.
- Measure what you produce.
- Strive for information rather than documentation.

It would be convenient to have one model to use in managing software development work.

Fig. 1 represents a minor redefinition of the traditional development process. Rather than aiming the development process toward producing lines of code, it can be viewed as producing a workable design for a problem. Notice that the traditional definition and the proposed definition are not incompatible. They are simply different ways of looking at the same activities.

The design process may be redefined as the specification of a design, which will either solve a stated problem or fulfill stated requirements. Traditionally, it was viewed as the specification of code that would perform a business function to solve the problem. The code and unit test activities may be viewed as the implementation of the design. Integration testing can be thought of as verifying that the implementation of the design actually meets stated requirements.

(A word about the difference between validation and verification. Validation ensures that an implementation of a design actually behaves as the design intended. Verification determines that a design has been consistently stated and constrained throughout its life cycle. In practice, verification should occur at every step in the life cycle. Most programmers confuse the two concepts but they always perform both kinds of tests.)

Unit tests normally make sure the program behaves the way the programmer intended it to behave. Integration and system tests ensure that all programmers used the same design specifications.

The redefining of the process has several appealing aspects. It allows new technology to enter software development. Purchased software and application generators no longer fall outside of the management model. The definition of requirements and the generation of specifications fall into the design process. Implementation simply replaces coding with, say, purchase of software or generation of application code. And validation of code may or may not be appropriate, depending upon circumstances.

Prototyping can be considered an alternate form of design expression. A prototype is a working model of the function, interfaces, and data required for a new system. If one considers the information contained in a prototype rather than the method of capturing the information, then prototyping falls into the design phase.

The redefinition of the phases focuses attention on meeting the needs of the business rather than on the act of creating a document or writing a program.

Using the new definition, software developers are now positioned to eliminate the verification step. Typically, integration and system testing will occupy 20% to 35% of the project development time. By focusing attention on the design, back-end testing may be no longer necessary. If, over a period of time, developers can demonstrate that their designs are consistent and complete prior to implementation and the implementation can be validated, then verification after implementation becomes redundant.

Errors in software will be reduced. Estimates of errors introduced in the specifications phase range from 40% to 60% of all software errors. Increased attention to the verification of design statements should significantly reduce the number of errors in the software.

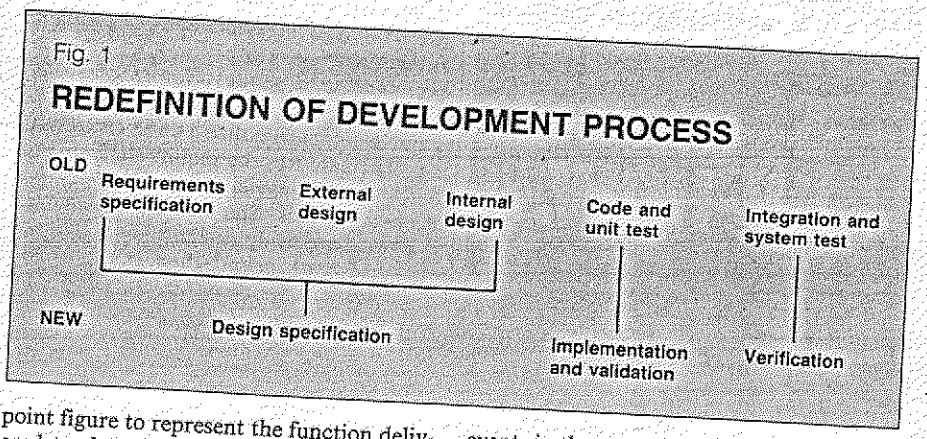
MEASURING DESIGN PROCESSES

Knowing what to measure is just as important. Measuring design processes is not very well understood. Any attempt will be experimental in nature. The best one can hope to do is to try several alternatives and evaluate which, if any, seem to make sense.

Hardware developers use measurements such as MIPS and bits per inch, which provide some standardized idea of how much utility is being delivered to the customer. Software developers don't have any standard measures of utility, nor do they mass-produce standard products. Some measures of delivered function have been attempted, notably function points. Although not universally accepted as a valid measurement, function points are the most widely known estimator of delivered function.

The disadvantage of function points is that they are imprecise and often misunderstood. Many people perceive a function

Fig. 1



point figure to represent the function delivered to the user. In fact, it represents the amount of function imbedded in the specific design of the system; much of the imbedded function may be invisible or not utilized by the user. Indeed, different designs meeting the same set of requirements may have widely different function point counts.

An approach to quantifying design content would be a design methodology that yields consistent designs. The data-oriented design methodologies like the Jackson and the Warnier-Orr methodologies can be used to derive consistent design data. Several people working on the same problem will develop very similar designs using the data flow techniques. Functional decomposition techniques often yield vastly different design approaches to the same problem.

The advantage of consistent design products is that they will all deliver the same level of detail and can be examined for complexity by counting the number of transformations. Thus, a design could be measured by quantifying the number of transformations and low-level elements it required. The disadvantage of the data-oriented design techniques is that they are unsuitable for some classes of problems.

The use of a design tool or a design language would simplify the problem. If a design is expressed in a consistent fashion, then some measure of its contents can be made. For example, PSL/PSA uses expressions about data objects, processes, inputs/outputs, and system structure to create a design database. A measure of interfaces, processes, and data at several hierarchic levels might be used to quantify the scope of a design. Completeness and consistency can be expressed in terms of mismatched interfaces and processes, or by the data a process uses. SADT (structured analysis and design technique) defines a system as a collection of objects and events. It allows any system to be described as a mapping between the objects of the system and the events in the system. Measures of the scope and consistency of the system could be derived from the SADT database.

DOZENS OF TOOLS AVAILABLE

There are dozens of languages and tools available. The tools not only enable the developer to collect data consistently, they allow him or her to manage the process over the period of the development process. Once the design is consistently expressed in the database, changes to the design can be consistently and completely implemented. Additions or enhancements can be applied and the database reviewed to determine their impact on other parts of the system. If a hierarchy of design detail is used, each level can be expanded into increasing detail and checked for completeness, e.g., for processes without inputs, data elements that are not used, and so forth. The expansion of requirements into design and design into detail design will provide some confidence that requirements can be traced to specific design features.

Measurements of the implementation process will make certain that the implemented (coded, purchased, or prototyped) product matches the lowest level of design specification. The process for performing these tests will vary depending upon the kind of specifications used. A data flow design can be validated by ensuring that every input/output mapping is correct. An event-driven design can be tested using the mapping of all specified triggers and events. If the design is specified in terms of logic sequences or flowcharts,then the test group may be reduced to attempting to test logical paths through the system.

Measurements of validation can be expressed in a relatively noncomplex manner if we simply make a statement about design specifications that were implemented incorrectly. An approach to implementing such a philosophy might be organized as follows.

- Coding milestones are the completion of major design points instead of the completion of programs or transactions.
- Unit test milestones are the validation of major design features.
- Errors are expressed in design features not delivered or delivered incorrectly.

This approach encourages programmers to be clear about the specifications they are trying to implement. This is important because we have noted that a large proportion of all errors are design errors. It also encourages the inspection/review process to concentrate upon meeting the design criteria rather than on critiquing coding styles. Finally, it provides a statement of errors that is of interest to the customer.

The customer is interested in a statement that the system works or that parts of it don't work. A measurement showing that there is one error per KLOC is unlikely to be of interest to a user except to make him wonder which lines of code aren't working. A statement that all critical interfaces work properly and that all minor errors are itemized is more likely to instill confidence in the customer. To be useful, a measure of the validation process must address the activity of validation, not some abstract concept of error density.

Measurements of the verification process should express some concept of the consistency of the design across the development process. Such a metric should at least attempt to correlate initial requirements statements and low-level design statements. One approach might be to identify user interfaces in relation to requirements statements. Another might be to show the expansion of design for each component in the requirements. Certainly the measure should ensure that the interfaces between components are stated consistently.

ORGANIZE TO REDUCE ERRORS

Any kind of comprehensive completeness or consistency checking would probably have to be automated for all but very simple designs. In the absence of a design tool to provide automated verifications, it is possible to organize the design to reduce opportunity for error.

- Hierarchy, modularity, and data independence make designs less complex and less affected by changes to specifications.
- Whenever possible, specify functions in terms of hierarchies of data flows, input-process-output, or similar constructs rather than control-logic flows.
- Define interfaces between components at each hierarchical level as soon as possible.
- Consider using a set of data access mod-

AMERICA'S MOST VALUABLE FOREIGN AID IS BUSINESS KNOW-HOW.

HELP US PASS IT ON.

Albert V. Casey
Chairman of the Board & CEO
American Airlines, Inc.

I'm a volunteer supporter of the International Executive Service Corps, a not-for-profit organization with a vital mission:

We send retired U.S. executives to help companies in developing countries. The executives are volunteers. We pay their expenses, but they receive no salary.

Our main purpose is to help developing countries succeed in business. But the benefit doesn't stop there. These countries consume about 40 percent of U.S. exports. So the work we do helps to create jobs and earnings right here in America.

The International Executive Service Corps has completed 8,500 projects in 72 countries. I think you should seriously consider supporting this effort with funds and personnel. You would be in good company. Over 800 U.S. companies have supported us. Our Board of Directors and Advisory Council include the chief executive officers of many of America's most important corporations.

When you think about corporate giving, think about doing good business, as well as doing good.

International Executive Service Corps

It's not just doing good. It's doing good business.

Ad Council
A Public Service of This Publication

Join me in helping businesses in developing countries—by supporting the International Executive Service Corps. For more information, write to Albert V. Casey, Chairman of the Board & CEO, American Airlines, Inc., at 8 Stamford Forum, P.O. Box 10005, Stamford, CT 06904-2005. Or simply call this number: **(203) 967-6000.**

Name_____

Address_____

City_____ State_____ Zip_____

ules rather than imbedding accesses in functional modules.

All of the preceding techniques will simplify the verification, either manual or automated, of the design package.

The industry seems to be unable to distinguish between the information needed to produce a product and the documentation that is simply regurgitated in response to a request for information. Rather than publish phase documents we might attempt to utilize a design methodology that captures the required information and makes it available in easily comprehended sections. Some development organizations are using design tools to capture the information and then dumping the database, editing the result, and adding minor prose sections instead of writing phase documents. In response to requests for specific information, it is possible to generate reports from the database. The requester can be sent a soft copy of the specific design information he requested instead of an entire document.

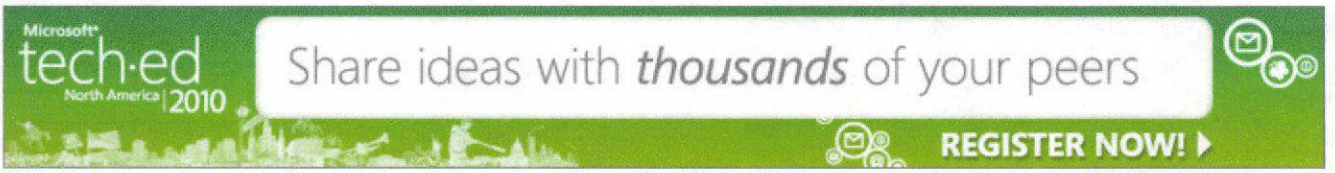
Control is a necessary part of all management processes. In fact, the phase development process came about as a way to control software development. But the cost of control functions is becoming disproportionately large in relation to the cost of development activities.

In general, avoid control-oriented tasks as much as possible. Try to produce control information as a result of normal activities. For example, by using a design database the number of people needed to draft and edit a review document is reduced. The database can be used to produce the design information, to write the prose, and edit the entire package. The use of a database eliminates the need for "writing" a design document.

At requirements time some kind of review ought to be held regarding the scope and anticipated cost of the project. During the design phase, there may need to be a review of the external interfaces and expected performance of the target system. If coding is to be done, then some sort of verification of the detail design is appropriate before beginning coding. Certainly a validation of the implemented system is necessary before it is delivered to the customer.

While management attention has wandered until control of the process has become more important than delivery of the system, the concept of the phase review process is still sound. But shifting attention from controlling the development process toward managing the development of a design will provide a clear understanding of the tasks and issues involved in the development process. The development process should provide an environment in which necessary control information can be generated without extra effort on the part of the developers. ◉

Patrick Brown is a programmer/analyst in the Information Systems Group at IBM.



Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

2002 Dr. Dobb's Excellence in Programming Awards

Since 1995, Dr. Dobb's Journal has presented its Excellence in Programming Award to individuals who, in the spirit of innovation and cooperation, have made significant contributions to the advancement of software development.Adele Goldberg and Dan Ingalls are pioneers in object-oriented programming in general, and the Smalltalk language in particular.

By Dr. Dobb's Journal
May 01, 2002
URL:http://www.ddj.com/184405043

Since 1995, *Dr. Dobb's Journal* has presented its Excellence in Programming Award to individuals who, in the spirit of innovation and cooperation, have made significant contributions to the advancement of software development. Past recipients of the Dr. Dobb's Excellence in Programming Award include:

- Alexander Stepanov, developer of the C++ Standard Template Library.
- Linus Torvalds, for launching Linux.
- Larry Wall, author of Perl.
- James Gosling, chief architect of Java.
- Ronald Rivest, educator, author, and cryptographer.
- Gary Kildall, for his work in operating systems, programming languages, and user interfaces.
- Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson, authors of *Design Patterns: Elements of Reusable Object-Oriented Software.*
- Guido van Rossum, Python creator.
- Donald Becker, for his contributions to Linux networking and the Beowulf Project.
- Jon Bentley, computer-science author and researcher.
- Anders Hejlsberg, developer of Turbo Pascal and architect of C# and the .NET Framework.

The recipients of this year's award, Adele Goldberg and Dan Ingalls, are pioneers in the area of object-oriented programming in general, and the Smalltalk language and development environment in particular. As researchers at Xerox's Palo Alto Research Center (PARC), Goldberg and Ingalls each recognized in their own way the promise of objects, and they were in a unique position to put those theories into practice in an architecture based

on objects at every level.

Adele Goldberg

Although we take objects for granted today, these two researchers helped to bring object-oriented programming into the real world for the first time almost 30 years ago, from the highest level of users and their information modeling needs to the lowest levels of syntax, compilation, and efficient message passing.

Looking back on the original work at Xerox, Goldberg later said it tackled one of the most difficult and problem-prone steps in software development — identifying terms and relationships as understood by human participants of a particular situation with those understood by a computer.

To that end, Goldberg believed that:

- Interactive, incremental software-development environments could produce a qualitative improvement in software-development productivity.
- Software could be designed in autonomous reusable units, each corresponding to identifiable entities (conceptual as well as physical) in the problem domain that communicate through well-defined interfaces.
- The model, or framework, for how these units work together represents both a process and vocabulary for talking about the problem domain.
- We should think about writing software in the context of building systems, rather than in the context of black box applications.

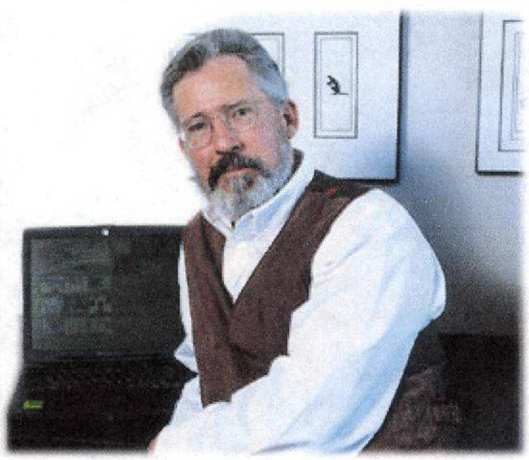
As early as 1977, Goldberg, along with Alan Kay, presented the goals for the Smalltalk research efforts in a paper entitled "Personal Dynamic Media" (*IEEE Computer,* March 1977). She went on to author and coauthor many of the definitive books on Smalltalk-80 programming including, with David Robson, the seminal *Smalltalk-80: The Language and Its Implementation* (Addison-Wesley, 1989, ISBN 0201136880), as well as numerous papers on object technology. Goldberg edited *The History of Personal Workstations* (ACM/Addison-Wesley, 1988; ISBN 0201112590); coedited with Margaret Burnett and Ted Lewis *Visual Object-Oriented Programming* (Prentice Hall, 1995; ISBN 0131723979); and coauthored with Kenneth Rubin *Succeeding with Objects: Decision Frameworks for Project Management* (Addison-Wesley, 1995; ISBN 0201628783).

Goldberg received her Ph.D. in Information Science from the University of Chicago for work carried out jointly at Stanford University. She also holds an honorary doctorate from the Open University (UK) in recognition of contributions to computer science education. After more than a decade as a researcher and laboratory manager at Xerox PARC, Goldberg became the founding CEO of ParcPlace Systems, the PARC spin-off that developed commercially available object-oriented application-development environments. Goldberg currently is founder of Neometron, a consulting company that focuses on dynamic knowledge management and support for project-based online communities.

From 1984 to 1986, Goldberg was president of the ACM, recipient of the 1987 ACM Systems Software Award along with Dan Ingalls and Alan Kay, and is an ACM Fellow. She received *PC Magazine*'s Lifetime Achievement Award in 1990.

Like Goldberg, Dan Ingalls was an original member of the PARC team that developed Smalltalk. He has been the principal architect of numerous Smalltalk virtual machines and kernel systems. The first of these, Smalltalk-72, supported the work reported in "Personal Dynamic Media." Smalltalk-76, described in ACM's 1978 Principles of Programming Languages (POPL) proceedings (and available at http://users.ipa.net/~dwighth

/smalltalk/St76/Smalltalk76ProgrammingSystem.html), was the first modern Smalltalk implementation with message syntax, compact compiled code, inheritance and efficient message execution, and its architecture endures in Smalltalk-80, the major documented release of Smalltalk work at Xerox. Most recently he designed the kernel of the Squeak open Smalltalk system, a practical Smalltalk written in itself. (For more information about Squeak, see ftp://st.cs.uiuc.edu/Smalltalk/Squeak/docs/OOPSLA.Squeak.html.) Ingalls also invented the BitBlt graphics primitive and pop-up menus, and was the principal designer of the Fabrik visual-programming environment while at Apple Computer.



Dan Ingalls

Ingalls received his Bachelor's degree in physics from Harvard University, and Masters in electrical engineering from Stanford University. He is a recipient of the ACM Grace Hopper Award and the ACM Software Systems Award. Ingalls currently works with Alan Kay and other seasoned Smalltalkers at Viewpoints Research Inc., where he is working to complete an architecture for modular Squeak content that is sharable over the Internet. He supports an active Squeak community (http://www.squeak.org/) through his participation in e-mail discussions, attention to periodic releases, and other support at all levels. He also runs Weather Dimensions (http://www.WeatherDimensions.com/), a company that sells a weather station he designed.

Although Goldberg and Ingalls worked at very different levels, the breadth of their collaborative territory is what shaped the final result. Ingalls says of his technical achievements, "I loved the challenge in efficiency and generality that it took to make Smalltalk real, but what gives me the most satisfaction looking back is that we built a serious system that is actually fun to use. We had a passion, inspired by Alan, to liberate the beauty of computer science from the barnacled past of ad hoc engineering." Goldberg adds, "During the PARC days, the opportunity to work with children and other nontechnical users kept us focused on how to use rigorously what people already know informally about objects. But the most thrilling experience for me was to work with ParcPlace customers in both large and small companies, and see how our technology enabled them to finally break the barrier between business understanding and systems implementation."

At Adele Goldberg's request and in her name, *Dr. Dobb's Journal* is pleased to make a grant of $1000 to the Girl's Middle School (http://www.girlsms.org/), a San Francisco Bay Area all-girls middle school that focuses on math and technology. At Dan Ingalls request and in his name, we are happy to make a $1000 grant to the The Sierra Nevada Children's Museum in Truckee, California. Please join us in honoring Adele Goldberg and Dan Ingalls who once again remind us that a mix of technology, innovation, vision, and cooperative spirit is fundamental to advancement in software development.

DDJ

&FS\UUKW II I I I 8 CMG%XVLCHVV0 HGDL/ / &

PREMIUM SEATING AVAILABLE NOW.
CLICK HERE FOR THE BEST SEATS IN THE HOUSE.
VISA

SOFTWARE MADE SIMPLE

Will Object-Oriented Programming Transform the Computer Industry?

While at engineering school, Eric Bergerson learned to write computer programs the hard way--line by bloody line. He would spend long nights tediously outlining and writing lists of instructions in C, a popular but rather touchy computer language. A single typing error could blow a program sky-high. And adding new functions, even to a smoothly running, well-understood program? That could take weeks or months to get right: Even the best-made programs were usually so convoluted that a seemingly trivial change could screw things up. "It was gnarly," he says.

Programming didn't get any easier when Bergerson went to Shearson Lehman Hutton Inc. in 1988. Only there was tons more pressure to do it fast. He began programming Sun Microsystems Inc. workstations for equity arbitrageurs and found that almost everything in those systems needed constant updating--from the details of transactions and trading strategies to the customized "look and feel" each trader wanted for his or her screen. Writing line after line of computer code, Bergerson hit all the same snags he had encountered at school. Compounding his frustration, he learned that down the hall in capital markets another young software hotshot, Alex Cone, was writing many of the same programs. Wasn't there a better way?

'LEGO BLOCKS.' There sure was. And as soon as Bergerson and Cone found it, they knew that for them--and someday, the rest of the world--programming would never be the same. Indeed, at the software startup they now head, Objective Technologies Inc., programming seems downright juvenile: Instead of mucking around in tangles of C code--writing arcane statements such as printf ("%s/n", curr str)--they mainly connect boxes on the screens of their NeXT Computer Inc. workstations and fill in blanks. In minutes, they have industrial-strength programs that run right the first time and that can be modified without brain surgery. Says Bergerson, 27: "I showed my mother, and she said, 'You're still playing with Lego blocks, like when you were a kid!' "

What they're doing is object-oriented programming. Some say it's just the latest computer buzzword, like artificial intelligence was a decade ago. They predict that like artificial intelligence, object-oriented programming will not spawn a distinct new set of products but will be a technique added to conventional software.

But unlike artificial intelligence, which promised the fascinating but far-out concept of computers that "think," object technology has an immediate, practical payoff. Already, it's helping the computer industry with its most daunting challenge: making software easier to create, simpler to use, and far more reliable.

That's a tall order. While computer hardware has made enormous strides, software has been largely mired in the past. Every two years or so, a new generation of microprocessor chips arrives and doubles hardware performance, but no such breakthrough has occurred in software. For the most part, programmers continue to cobble together software at a painfully slow rate. As a result, corporate programming departments are frequently a year or more behind. And computer makers and software suppliers often miss software shipment dates by months.

BIG PLANS. The bottom line: For lack of software, many of the advances in computer hardware go untapped. The software gap--yawning wider every year--is one reason for slow growth in computer sales. Object programming, however, "will get the industry out of the rut we're in," says Philippe Kahn, president of Borland International Inc.

Kahn and object technology's many other boosters predict that it will do for

software what the microchip has done for hardware. Instead of microchips, the software revolution will be built on so-called objects--simple, self-contained, reliable software components (box, page 92 58). Like the microprocessor, object technology has the potential to radically change the economics of the business--and not just in the $30 billion packaged-software industry. In an era when hardware is a commodity and software is the key competitive technology, computer makers that exploit object-oriented software best are likely to dominate the computer industry itself.

If you doubt that, consider the pending collaboration between IBM and Apple Computer Inc. These blood rivals stunned the industry last summer by announcing that they will work together. Their plans remain sketchy, but a key goal will be to create a system for object-oriented programming that will set a standard in the next decade--and thereby seize control of the industry from Microsoft Corp.

Object-oriented technology also figures prominently in the plans of William H. Gates III, Microsoft's chairman. In his view, every image, graph, or snippet of a road map will be stored in the computer as an object. The goal, says Gates, is "information at your fingertips"--the ability to seek out, compile, and summarize information from myriad electronic sources without having to know where any of it comes from.

Hoping to lead yet another technology movement, Steven P. Jobs has been pursuing object-oriented technology ever since he launched NeXT Inc. The NeXT workstation, introduced three years ago, comes complete with an object-oriented programming language and a library of 100 objects that handle such common tasks as printing, displaying information in windows, and handling electronic mail. It has become a favorite among software developers. Object programming, says Jobs, "is the first real technological shift we've had in the industry since the Macintosh."

INFINITELY REUSABLE. The key breakthrough in object technology is the ability to build large programs from lots of small, prefabricated ones. That's possible because objects completely change the traditional relationship between programs and data, which have been strictly segregated for 40 years. As the old term "data processing" implies, programs ordinarily act on data--simple lists of numbers or customer names, for example. An object, in contrast, encapsulates programs and data in one self-contained unit, which fully describes some real-world entity.

Think of the way an Apple Macintosh handles a page of information. The page on the screen is a rudimentary object. It has data--words, numbers, and graphs--and also the programming that lets it behave like a real page. Using your mouse, you can pick it up, move it, file it, copy it, or even throw it away.

This simple idea provides tremendous benefits. Software objects can be built to represent just about anything--from an abstract concept, such as an insurance policy, to a specific thing or person, such as Duke Ellington, American composer and musician, 1899-1974. More important, objects can be created that perform certain common tasks--sorting, for example. Once perfected, such objects are infinitely reusable, so programmers don't have to reinvent the wheel every time. Brad Cox, who created Objective C, the programming language that comes with NeXT machines, predicts that object technology will be as big an advance for the Information Age as Eli Whitney's invention of interchangeable musket parts was in the Industrial Age.

But software components are more than interchangeable cogs. Because they're made of programming and data, they "know" what they are and how they behave. An object called Payday, for instance, can automatically check with an object called Employee Roster, note any resignations or retirements, then call over to another object called Payroll and give it a list of checks to print--all without human intervention.Using reusable blocks, instead of writing from scratch, makes programming far faster and produces finished software that is more reliable and easier to update. Reusability alone is expected to give businesses a huge boost in programmer productivity (box) because eventually, only unique new functions will need to be written from scratch. Modifying programs is also easier. When NeXT wanted to give its workstations the ability to send faxes, for instance, it didn't have to write fax code into each program. It just added the fax programming to the workstations' Print object. Since all NeXT programs use that object, they were all instantly upgraded to communicate by fax.

For ordinary computer users, objects mean PCs that are far easier to use than today's most "user-friendly" machines. Indeed, when Xerox Corp.'s Palo Alto Research Center (PARC) began looking into object-oriented software in the 1970s, one of its goals, literally, was to design a system so simple a child could use it. Twenty years later, object-based technology promises to make computers easy enough for adults to use. "If my 5-year-old kid can use it, I consider it good," says Bjarne Stroustrup, an

AT&T Bell Laboratories computer scientist who invented the most popular object programming language, C++.

A good example of how objects can make PCs easier to operate is multimedia software, which gives computers the ability to manipulate snippets of video and sound. In a package called Macromind Director, by pointing to an icon that represents a VCR, you can retrieve still pictures or even film clips from computer files. The VCR object works much like the real thing: Select the "cassette" with the images you want, hit rewind or fast forward, and locate, say, a clip of the Hindenburg crash. Hit record, and copy the clip into your quarterly earnings presentation.

LIKE LIFE. Such multimedia tricks are only the glitzy surface of object-oriented programming. A more intriguing possibility is software that does a much better job of simulating how a business works than spreadsheets and data bases can. Businesspeople "want to describe information in more general, real-world terms and create a full simulation of what they think is going on," says Adele Goldberg, a former Xerox PARC researcher and now president of Parc-Place Systems, a maker of object-based software.

Three years after Bergerson and Cone left, Shearson Lehman Brothers Inc. has bought into object technology and is building software that simulates its business. It has Account objects, representing customers, Contract objects to manage agreements between parties, and Security objects that describe the properties of stocks, bonds, or options. An Account can enter into a Contract to buy a Security--just as in life. "It's more toward the reality of what's actually happening," observes Shearson Vice-President Frank Filippis.

Once objects have been built and tested, it's fairly simple to clone them for new products or services. Now, when Lehman wants to sell a new type of security, the programmers just tell the computer the special attributes of this new instrument. The Security object then automatically gives birth to a program that inherits all its generalized traits, plus the unique new attributes. "We can model all types of securities this way," says Filippis.

The upshot is a system that can keep up with business changes. At Unum Life Insurance Co., for example, whenever a state regulation changed in the past, programmers for the Portland (Me.)-based insurer had to scramble. But now, using objects, they can do such updates in one-third the time--and create software that's far more usable by nontechies, says Barby Muller, a technology manager. In some cases, "instead of the programmers, the business people can make changes to the software," she says.

Another big benefit: By building programs from prefab objects, you avoid the kind of "spaghetti" code that programmers commonly use to patch new functions onto old systems. These little programs can make software maintenance--usually the biggest cost in running a computer center--a nightmare. Brooklyn Union Gas Co. recently scrapped a 13-year-old customer information system on its mainframe that had become so huge and inflexible that the company couldn't respond to the needs of its 1 million customers. With Andersen Consulting, it created an object-based program that's 40% smaller yet does more. And the company expects it to last 20 years--on a fraction of the old maintenance budget.

Such success stories are attracting more converts. In a recent survey by researcher International Data Corp., 70% of large U. S. corporations said they are programming with objects or plan to do so soon. The main motivator? Money. Shearson's Filippis claims that his group has cut 30% from development costs. He reckons the company could save millions more if every department shared a central object library.

Hard to imagine that a single technical advance can do all this--drastically improve programmer productivity, create more reliable software, and give computers a childlike simplicity? Surely, there must be a catch. There are several.

Among the most formidable: It takes a lot of careful planning to create objects. Software designers not only have to figure out what each building block should do but they also must anticipate how each will work with thousands of other objects. "It takes a lot of engineering to make things look simple and easy," warns Stroustrup of Bell Labs.

PROJECT PINK. An even bigger obstacle may be standards. The big payoff from object-oriented software will come when there are common ways to shuttle objects between different computers. To that end, more than 160 computer and software makers and customers have joined the Object Management Group. Its goal is to create an electronic system to distribute software objects, such as multimedia documents, across a network, regardless of the type of computers that are on it. Digital Equipment, Sun, and Hewlett-Packard are now collaborating to produce the software. Meanwhile, the Apple-IBM camp is working furiously to create a standards-

setting object-oriented operating system--the basic program that runs a computer. That effort, say industry-watchers, will be based on Pink, an object-oriented system under development at Apple. It also will include technology that IBM acquired with the purchase of Metaphor Computer Systems, a software company headed by David E. Liddle, another Xerox PARC alumnus.

Companies that will compete with the IBM-Apple alliance--Sun Microsystems, Microsoft, and NeXT--argue that it's not necessary to build an all-new operating system to deliver the benefits of object-based software. "That's not a very realistic scenario," says Gates, who plans to slowly add object-based technology to Microsoft's operating systems.

'BLOATWARE.' Liddle contends that without an object-oriented operating system, customers won't realize the efficiencies inherent in the new technology. Worse, they'll be stuck with poor applications programs--what he calls "bloatware." These are aging packages to which hundreds of features have been added to make them "new" and "improved"--but almost impossible to master. A better idea, he says, would be to make nifty new features freestanding objects, easily accessed by any program. Such common objects may even be included with the operating system that IBM and Apple are building.

Eventually, a whole new way of selling software may emerge. In a market of interchangeable, plug-and-play objects, you might shop for pieces separately and compile your own custom software. Chunks of programs may be sold like hardware components. "You can walk into a Radio Shack and buy a chip or circuit that does a specific function," says Chuck Duff, founder of Whitewater Group, which makes programs to write object-oriented software. "That needs to happen for software."

How quickly object technology will sweep the industry is anybody's guess. Certainly, it's catching on with software makers and big corporations, who hope to make programming simpler and cheaper. But will it fundamentally alter the computer business, as some observers predict? Maybe. "The entire software environment needs a face-lift," notes Edward J. Zander, president of Sun's SunSoft subsidiary. Object technology by itself may not be the cure to slow growth, but it looks like a good bet for painting a happier face on the computer industry.

WHAT IS AN OBJECT?

Software objects are chunks of programming and data that can behave like things in the real world. On an Apple Macintosh computer, for example, you can use electronic objects called file folders and file cabinets to organize pages of information--the way you would in the physical world.

But objects can be applied to many kinds of programs. An object can be a business form, an insurance policy--or even an auto axle. The axle object would include data describing its physical dimensions--and programming that describes how it interacts with other parts, such as wheels and struts.

A system for a human resources department would have objects called employees, which would have data about each worker and the programming needed to calculate salary raises and vacation pay, sign up dependents for benefits, and make payroll deductions. Because objects have "intelligence"--they know what they are and what they can and can't do--objects can automatically carry out tasks such as calling into another computer, perhaps to update a file when an employee is promoted.

The biggest advantage is that objects can be reused in different programs. The object in an electronic-mail program that places messages in alphabetical order can also be used to alphabetize invoices. Thus, programs can be built from prefabricated, pretested building blocks in a fraction of the time it would take to build them from scratch. Programs can be upgraded by simply adding new objects.

John W. Verity and Evan I. Schwartz in New York, with bureau reports

REGISTER | BW HOME | BW CONTENTS | BW PLUS! | BW DAILY | SEARCH | CONTACT US

BusinessWeek

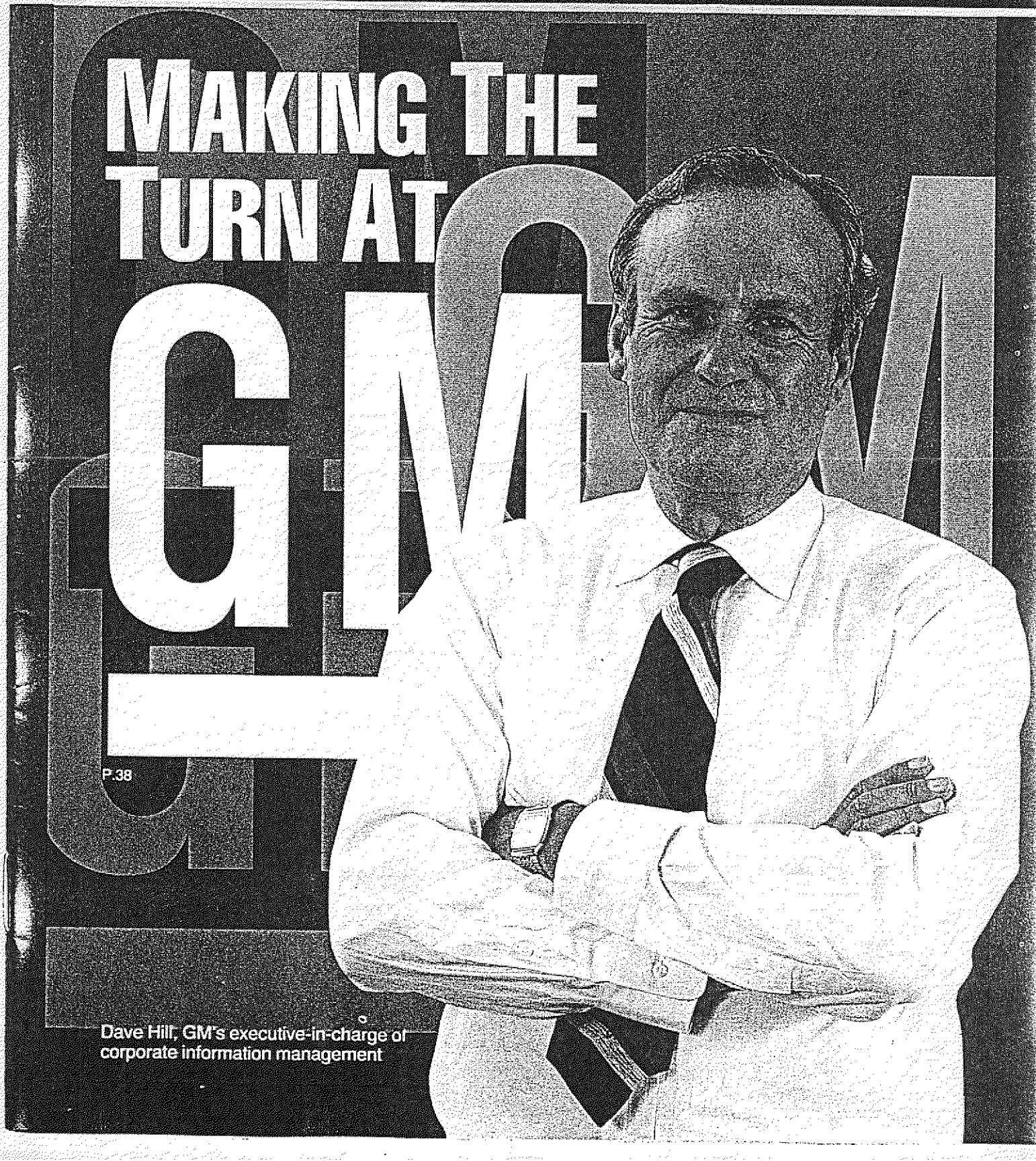
Updated Aug. 25, 1997 by bwwebmaster
Copyright 1991, Bloomberg L.P.

Zale Is Blue Jewel P.12 Businessland: What's In Store? P.13 Managing Less P.44

INFORMATIONWEEK
THE NEWSMAGAZINE FOR INFORMATION MANAGEMENT

MAKING THE TURN AT GM

P.38



Dave Hill, GM's executive-in-charge of corporate information management

THE OBJECT OF TI'S DESIRE

Object-oriented technology helps create a new production process

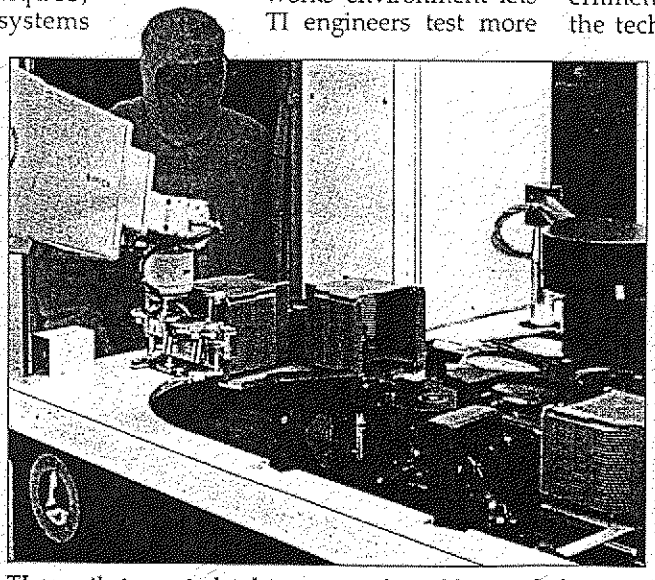
A leading semiconductor manufacturer and the U.S. Department of Defense are using one innovative technology to develop another. Texas Instruments Inc. is employing the object-oriented software development techniques used to build manufacturing support systems to design an entirely new production process.

"The approach we have taken is to adopt object-oriented programming to all aspects of the system," reports John McGehee, chief software architect for the micro electronics manufacturing science and technology project at TI. McGehee helps oversee one of the most ambitious semiconductor manufacturing projects ever attempted. The project, which is funded through a $112 million Defense Department contract won by TI in 1988, seeks to build a radically new semiconductor manufacturing facility capable of producing custom military chips in far less time and for far less money than is currently possible.

Object-oriented techniques, which attempt to describe systems as objects, are widely used to develop individual applications. But James Rumbaugh, a computer scientist at General Electric Co.'s R&D center in Schenectady, N.Y., and one of the creators of a high-level development methodology called the Object Modeling Technique, claims that object-oriented techniques are ideally suited for designing factories, distribution routes, and other businesses. "It's a natural way to model a real thing like a factory or business because it corresponds to the way people think about them," he explains.

As opposed to traditional software development methodologies, which list a sequence of operations to be performed, including fetching the appropriate data, object-orientation begins by representing the data as objects. The definition of those objects (i.e. buildings, cars, machinery, etc.) also includes operations or tasks that are associated with them (i.e. opening a door or closing a window). Therefore, a modular system comprised of objects includes both the data and the instructions acting on that data.

At Texas Instruments, McGehee and his colleagues have used object-orientation to simulate everything from inventory management to process control with Mountain View, Calif.-based ParcPlace Systems Inc.'s Objectworks for Smalltalk, an object-oriented development environment based on Unix workstations. Actual objects simulated at TI include a robot, a chip in production, and a process-control operator. The Objectworks environment lets TI engineers test more than 30 new processes that included the interaction between a variety of sensors and machines in just four months of prototyping. "Software had a significant role to play," says McGehee. "The simulation system is now viewed as the glue holding this all together."



TI says that new technology can produce chips much faster

The object-oriented approach has enabled TI to undertake numerous revisions and modifications to the production process that would have been prohibitively expensive using traditional development techniques. Unlike most chip plants, which churn out large quantities of chips with little variation, the TI project seeks to produce custom chips in smaller batches at low cost, using flexible manufacturing techniques. The Defense Department, which hopes the project will help reduce its dependence on foreign suppliers for high-tech weapons, ultimately plans to transfer the chip-making technology to U.S. semiconductor companies for commercial use.

Jim Feldhan, senior VP at Instat Inc., a Scottsdale, Ariz., market research firm, says a breakthrough of this sort would have a big impact on the semiconductor industry, since custom designs are the fastest growing part of the market. "Newer products tend to be lower volume anyway, until the technology proves itself," he says, "so if [TI and the Defense Department] could make it work it could be extremely beneficial to U.S. suppliers."

McGehee believes the U.S. government will succeed in transferring the technology to non-defense chip makers, in large part because the object-oriented systems underpinning the TI project are easily revisable. "We are viewing it as one of the most important and strategic programs in the country in terms of keeping the U.S. semiconductor industry at the forefront of technology," he says. "One message I would like to see propagated is that the object-oriented revolution has already begun. We have seen the benefits. The paradigm is extremely powerful."

—Will McClatchy

THE ECONOMY WILL LOWER RATES WORK?
PAGE 20

STRATEGIES: ▶CHRYSLER ▶INTEL ▶HEINZ

BusinessWeek

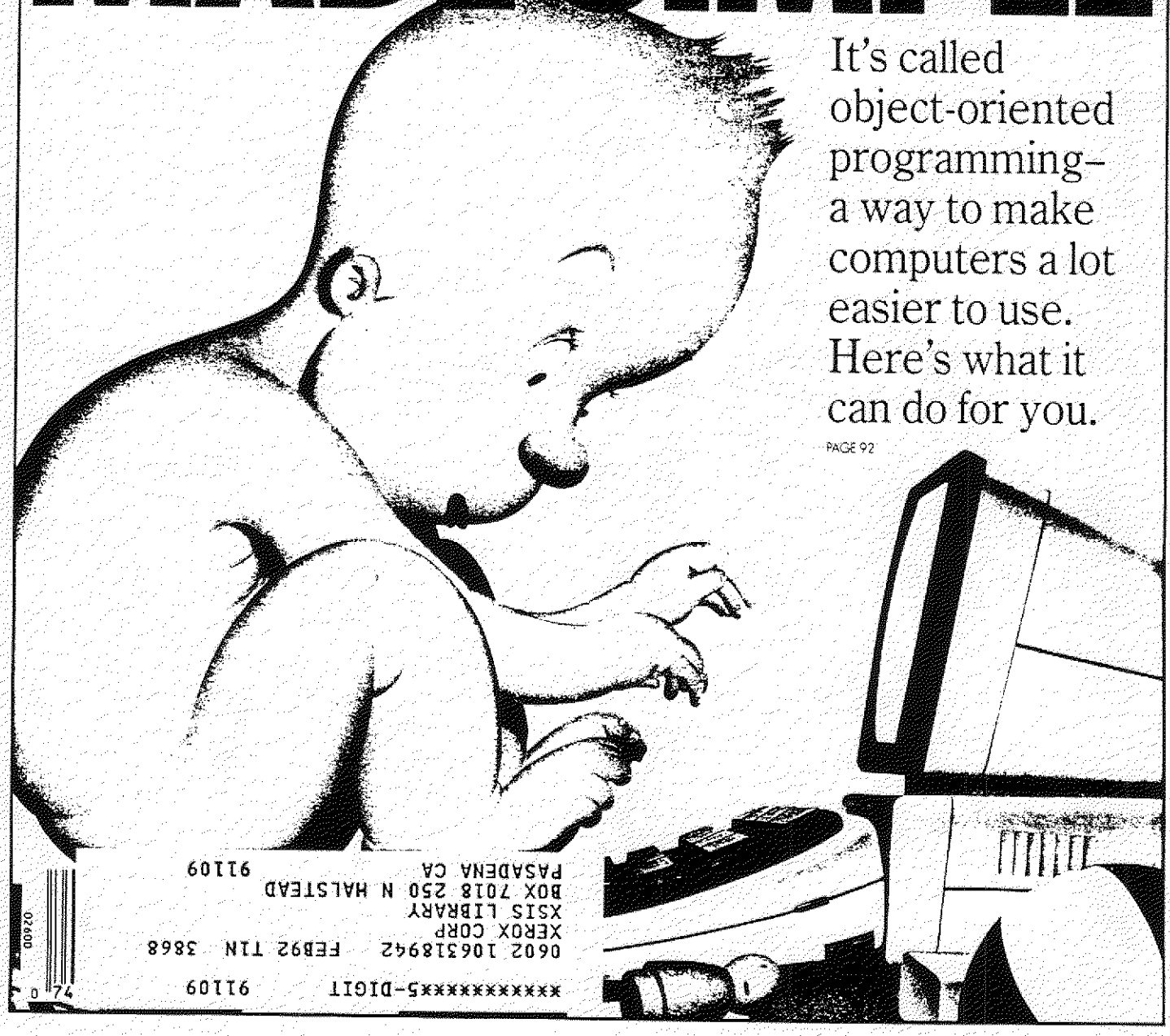
SEPTEMBER 30, 1991 A McGRAW-HILL PUBLICATION $2.50

$2128 29 30 31

SOFTWARE MADE SIMPLE

It's called object-oriented programming– a way to make computers a lot easier to use. Here's what it can do for you.

PAGE 92



0200
0 74
91109 PASADENA CA
BOX 7018 250 N HALSTEAD
XSIS LIBRARY
XEROX CORP
0602 106318942 FEB92 TIN 3868
*********5-DIGIT 91109

SOFTWARE MADE SIMPLE

WILL OBJECT-ORIENTED PROGRAMMING TRANSFORM THE COMPUTER INDUSTRY?

While at engineering school, Eric Bergerson learned to write computer programs the hard way—line by bloody line. He would spend long nights tediously outlining and writing lists of instructions in C, a popular but rather touchy computer language. A single typing error could blow a program sky-high. And adding new functions, even to a smoothly running, well-understood program? That could take weeks or months to get right: Even the best-made programs were usually so convoluted that a seemingly trivial change could screw things up. "It was gnarly," he says.

Programming didn't get any easier when Bergerson went to Shearson Lehman Hutton Inc. in 1988. Only there was tons more pressure to do it fast. He began programming Sun Microsystems Inc. workstations for equity arbitrageurs and found that almost everything in those systems needed constant updating—from the details of transactions and trading strategies to the customized "look and feel" each trader wanted for his or her screen. Writing line after line of computer code, Bergerson hit all the same snags he had encountered at school. Compounding his frustration, he learned that down the hall in capital markets another young software hotshot, Alex Cone, was writing many of the same programs. Wasn't there a better way?

'LEGO BLOCKS.' There sure was. And as soon as Bergerson and Cone found it, they knew that for them—and someday, the rest of the world—programming would never be the same. Indeed, at the software startup they now head, Objective Technologies Inc., programming seems downright juvenile: Instead of mucking around in tangles of C code—writing arcane statements such as *printf ("%s/n", curr str)*—they mainly connect boxes on the screens of their NeXT Computer Inc. workstations and fill in blanks. In minutes, they have industrial-strength programs that run right the first time and that can be modified without brain surgery. Says Bergerson, 27: "I showed my mother, and she said, 'You're still playing with Lego blocks, like when you were a kid!' "

What they're doing is object-oriented programming. Some say it's just the latest computer buzzword, like artificial intelligence was a decade ago. They predict that like artificial intelligence, object-oriented programming will not spawn a distinct new set of products but will be a technique added to conventional software.

But unlike artificial intelligence, which promised the fascinating but far-out concept of computers that "think," object technology has an immediate, practical payoff. Already, it's helping the computer industry with its most daunting challenge: making software easier to create, simpler to use, and far more reliable.

That's a tall order. While computer hardware has made enormous strides, software has been largely mired in the past. Every two years or so, a new generation of microprocessor chips arrives and doubles hardware performance, but no such breakthrough has occurred in software. For the most part, programmers continue to cobble together software at a painfully slow rate. As a result, corporate programming departments are frequently a year or more behind. And computer makers and software suppliers often miss software shipment dates by months.

BIG PLANS. The bottom line: For lack of software, many of the advances in computer hardware go untapped. The software gap—yawning wider every year—is one reason for slow growth in computer sales. Object programming, however, "will get the industry out of the rut we're in," says Philippe Kahn, president of Borland International Inc.

Kahn and object technology's many other boosters predict that it will do for software what the microchip has

WHAT IS AN OBJECT?

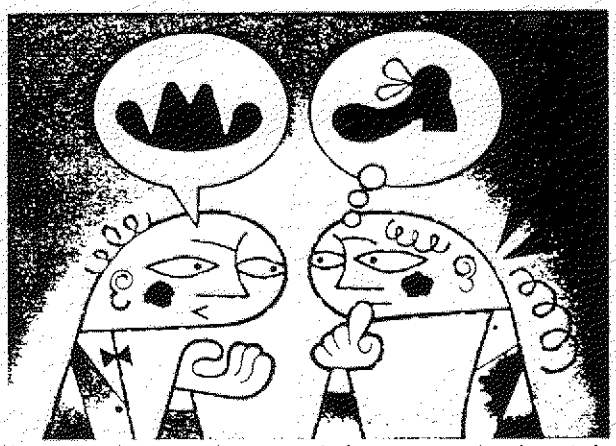
Software objects are chunks of programming and data that can behave like things in the real world. On an Apple Macintosh computer, for example, you can use electronic objects called file folders and file cabinets to organize pages of information—the way you would in the physical world.

But objects can be applied to many kinds of programs. An object can be a business form, an insurance policy—or even an auto axle. The axle object would include data describing its physical dimensions—and programming that describes how it interacts with other parts, such as wheels and struts.

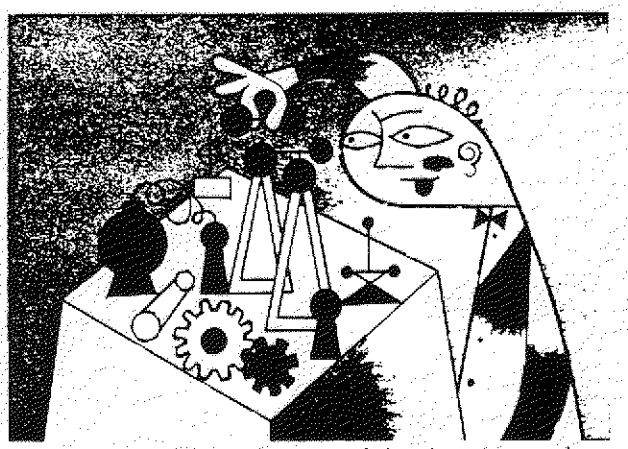
A system for a human resources department would have objects called employees, which would have data about each worker and the programming needed to calculate salary raises and vacation pay, sign up dependents for benefits, and make payroll deductions. Because objects have "intelligence"—they know what they are and what they can and can't do—objects can automatically carry out tasks such as calling into another computer, perhaps to update a file when an employee is promoted.

The biggest advantage is that objects can be reused in different programs. The object in an electronic-mail program that places messages in alphabetical order can also be used to alphabetize invoices. Thus, programs can be built from prefabricated, pretested building blocks in a fraction of the time it would take to build them from scratch. Programs can be upgraded by simply adding new objects.

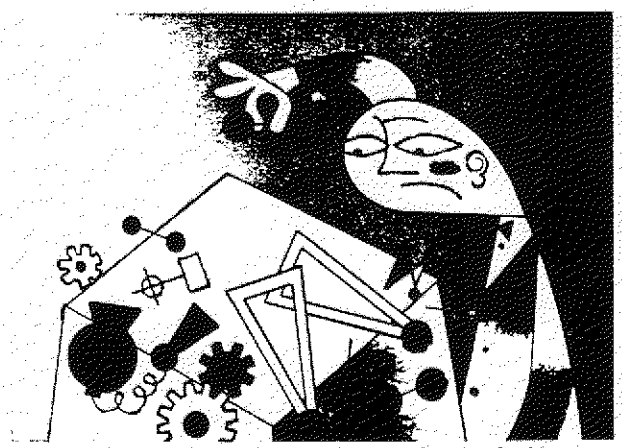
THE OLD WAY...



CONFUSION can begin even before programming does. As in the party game Telephone, the description of what a program should do is retold many times. By the time it's translated into a series of commands that the computer "understands," the original idea is easily distorted.

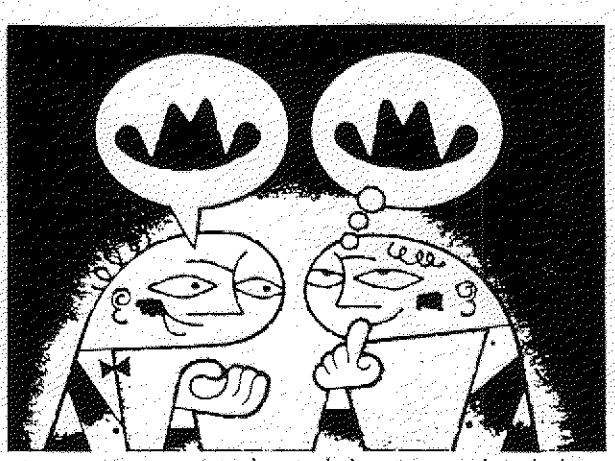


HANDCRAFTING programs can help wring extra speed from a system. But, usually, such one-off work can't be reused in other programs. That has kept software writing a quirky craft, rather than the modern manufacturing process it should be.

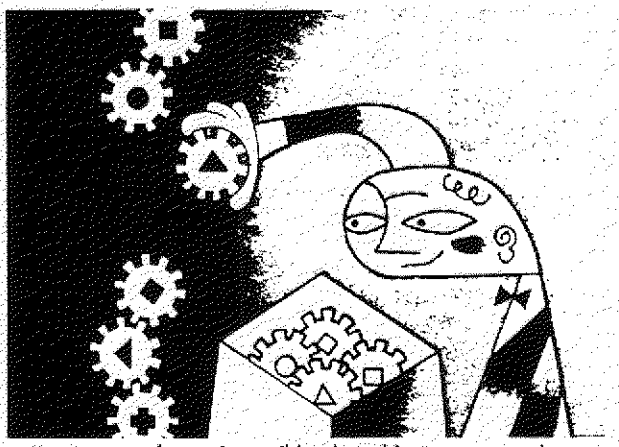


BREAKDOWNS can be a disaster with handcrafted programs. Often, only the programmer who wrote it knows how it works. Worse, "spaghetti code" can snake through a system, so altering one part of a program can have disastrous results elsewhere.

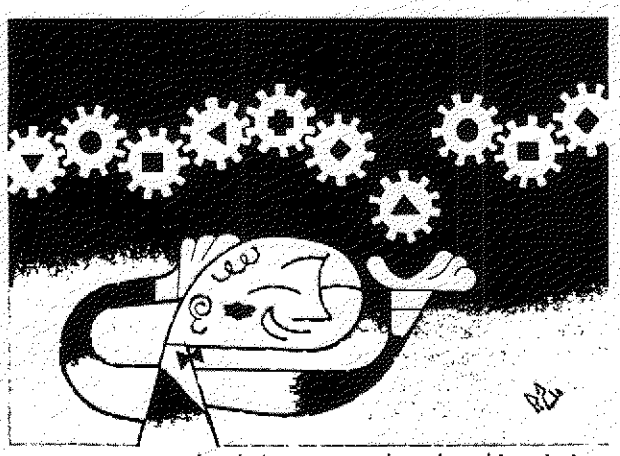
...THE NEW WAY



UNDERSTANDING how to design a program is easier because objects can correspond to real-world entities. If management wants to automate order taking, an order-slip object can be written to replicate the real thing, with spaces for address, quantities, prices, etc.



REUSING software is possible when objects are created according to precise rules. As long as objects conform to standards for how they should communicate and interact with one another, it's possible to reuse old, reliable objects in new programs. That saves time and money.



REPAIRING and updating programs based on objects is simple. Objects isolate program functions from each other, so that a change in one doesn't disrupt the program elsewhere. Entire objects can be swapped out for new ones, without reworking the rest of the program.

PHILIP ANDERSON

'IF MY 5-YEAR-OLD KID CAN USE IT, I CONSIDER IT GOOD'
BJARNE STROUSTRUP AT&T BELL LABORATORIES

done for hardware. Instead of microchips, the software revolution will be built on so-called objects—simple, self-contained, reliable software components (box, page 92). Like the microprocessor, object technology has the potential to radically change the economics of the business—and not just in the $30 billion packaged-software industry. In an era when hardware is a commodity and software is the key competitive technology, computer makers that exploit object-oriented software best are likely to dominate the computer industry itself.

If you doubt that, consider the pending collaboration between IBM and Apple Computer Inc. These blood rivals stunned the industry last summer by announcing that they will work together. Their plans remain sketchy, but a key goal will be to create a system for object-oriented programming that will set a standard in the next decade—and thereby seize control of the industry from Microsoft Corp.

Object-oriented technology also figures prominently in the plans of William H. Gates III, Microsoft's chairman. In his view, every image, graph, or snippet of a road map will be stored in the computer as an object. The goal, says Gates, is "information at your fingertips"—the ability to seek out, compile, and summarize information from myriad electronic souces without having to know where any of it comes from.

Hoping to lead yet another technology movement, Steven P. Jobs has been pursuing object-oriented technology ever since he launched NeXT Inc. The NeXT workstation, introduced three years ago, comes complete with an object-oriented programming language and a library of 100 objects that handle such common tasks as printing, displaying information in windows, and handling electronic mail. It has become a favorite among software developers. Object programming, says Jobs, "is the first real technological shift we've had in the industry since the Macintosh."

INFINITELY REUSABLE. The key breakthrough in object technology is the ability to build large programs from lots of small, prefabricated ones. That's possible because objects completely change the traditional relationship between programs and data, which have been strictly segregated for 40 years. As the old term "data processing" implies, programs ordinarily act on data—simple lists of numbers or customer names, for example. An object, in contrast, encapsulates programs and data in one self-contained unit, which fully describes some real-world entity.

Think of the way an Apple Macintosh handles a page of information. The page on the screen is a rudimentary object. It has data—words, numbers, and graphs—and also the programming that lets it behave like a real page. Using your mouse, you can pick it up, move it, file it, copy it, or even throw it away.

This simple idea provides tremendous benefits. Software objects can be built to represent just about anything—from an abstract concept, such as an insurance policy, to a specific thing or person, such as Duke Ellington, American composer and musician, 1899-1974. More important, objects can be created that perform certain common tasks—sorting, for example. Once perfected, such objects are infinitely reusable, so programmers don't have to reinvent the wheel every time. Brad Cox, who created Objective C, the programming language that comes with NeXT machines, predicts that object technology will be as big an advance for the Information Age as Eli Whitney's invention of interchangeable musket parts was in the Industrial Age.

But software components are more than interchangeable cogs. Because they're made of programming and data, they "know" what they are and how they behave. An object called Payday, for instance, can automatically check with an object called Employee Roster, note any resignations or retirements, then call over to another object called Payroll and give it a list of checks to

'THE FIRST REAL TECHNOLOGICAL SHIFT SINCE THE MACINTOSH'
STEVEN P. JOBS NeXT INC.

[TOP TO BOTTOM] PHOTOGRAPHS BY JEFFREY LOWE, ROBERT HOLMGREN

print—all without human intervention.

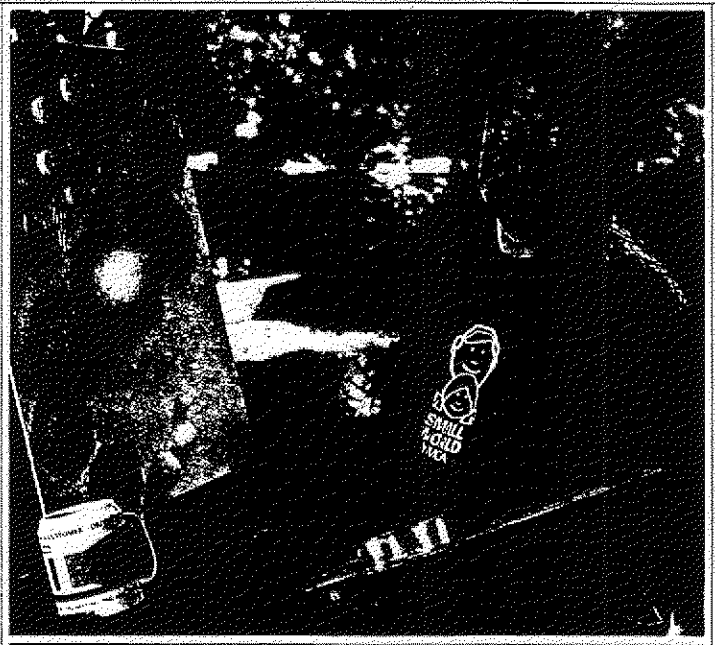
Using reusable blocks, instead of writing from scratch, makes programming far faster and produces finished software that is more reliable and easier to update. Reusability alone is expected to give businesses a huge boost in programmer productivity (box) because eventually, only unique new functions will need to be written from scratch. Modifying programs is also easier. When NeXT wanted to give its workstations the ability to send faxes, for instance, it didn't have to write fax code into each program. It just added the fax programming to the workstations' Print object. Since all NeXT programs use that object, they were all instantly upgraded to communicate by fax.

For ordinary computer users, objects mean PCs that are far easier to use than today's most "user-friendly" machines. Indeed, when Xerox Corp.'s Palo Alto Research Center (PARC) began looking into object-oriented software in the 1970s, one of its goals, literally, was to design a system so simple a child could use it. Twenty years later, object-based technology promises to make computers easy enough for adults to use. "If my 5-year-old kid can use it, I consider it good," says Bjarne Stroustrup, an AT&T Bell Laboratories computer scientist who invented the most popular object programming language, C++.

A good example of how objects can

AT HP THESE DAYS, OLD SOFTWARE NEVER DIES

As a $13 billion-a-year maker of computers, laser printers, calculators, medical systems, and electronic test gear, Hewlett-Packard Co. produces gobs of software every year. About 60% of its research and development funds and personnel are devoted to programming and improving the software-creation process. Several years ago, HP's top engineers realized that they could get a tremendous productivity boost if they could somehow reuse old chunks of software in new products—thus reducing the need to write new software from scratch for every new computer or heart monitor. Since then, a software revolution has been quietly brewing at the Silicon Valley giant.



'WE'RE NOT DRIVING PEOPLE TO USE OBJECTS. WE'RE TAKING ONE BITE AT A TIME'
MARTIN GRISS HEWLETT-PACKARD CO.

The leader of the movement is Martin Griss, a cherubic South African who has proclaimed himself HP's "reuse rabbi." He figures that if HP really gets serious about recycling its software, the company can save a cool $100 million annually.

QUALITY AND SAVINGS. So, when he's not pursuing his hobby, what he calls "object-oriented painting," Griss spends much of his time shuttling among HP's many software facilities around the world. There, he encourages engineers to consider reusing software at the start of every programming project. That means looking for useful chunks of software that already exist in other parts of HP and designing new bits of software in such a way that others can easily use them. Programming groups can try whatever techniques they want, but Griss advocates a gradual shift to object-oriented methods, because they offer the greatest potential for reuse. "We're not driving people to use objects," says Griss. "We're taking one bite at object-oriented [programming] at a time."

The reuse message seems to be getting through. One good example is a massive manufacturing program that helps HP customers keep tabs on their inventory and factory operations. Griss helped persuade programmers in four different HP divisions to swap preused software with each other instead of creating everything from scratch. Turns out the programmers were able to take 40% of their software from existing programs. That translates into savings of 15% in development costs, says Griss. And, because used software doesn't need as much tinkering, he estimates that maintenance costs will be less than half what they would be for virgin code. Better still, the quality went up—from four defects per 1,000 lines of code to only four per 10,000 lines.

Another standout success is CareVue 9000, a network of workstations that helps nurses record and manage patient information. HP's programmers faced an enormously complex task: The system would have to anticipate every hospital's unique record-keeping and medical procedures. "The last thing [the system] should do is dictate one way of doing things," says Robert Seliger, system architect at HP's clinical-information-systems operation.

'ROLL THEIR OWN.' So his team chose an object-based design that would let each hospital mold and extend CareVue to its specific needs. Objects "let the hospitals roll their own" software, Seliber says. For example, they can create data-entry forms that look just like the paper ones their nurses have always used. But the electronic objects work better because the program can easily adapt to the types of treatments in which each hospital specializes.

Despite these successes, Griss says that he still often encounters reluctance among HP engineering groups to buy his message. "The impediments," he says, "are social more than technical." Engineers sometimes feel they should be paid more for the extra work that's needed to make software modules that can be used by others. Moreover, they often don't think to look for prewritten components until it's too late. But Griss is there, as he puts it, "to let people know there's good stuff in the library." A reuse rabbi's work is never done.

By John W. Verity in New York

JOHN HARDING

THE TECHNOLOGY IS AN ADVANCE AKIN TO THAT OF INTERCHANGEABLE MUSKET PARTS
BRAD COX OBJECTIVE C CREATOR

make PCs easier to operate is multimedia software, which gives computers the ability to manipulate snippets of video and sound. In a package called Macromind Director, by pointing to an icon that represents a VCR, you can retrieve still pictures or even film clips from computer files. The VCR object works much like the real thing: Select the "cassette" with the images you want, hit rewind or fast forward, and locate, say, a clip of the Hindenburg crash. Hit record, and copy the clip into your quarterly earnings presentation.

LIKE LIFE. Such multimedia tricks are only the glitzy surface of object-oriented programming. A more intriguing possibility is software that does a much better job of simulating how a business works than spreadsheets and data bases can. Businesspeople "want to describe information in more general, real-world terms and create a full simulation of what they think is going on," says Adele Goldberg, a former Xerox PARC researcher and now president of ParcPlace Systems, a maker of object-based software.

Three years after Bergerson and Cone left, Shearson Lehman Brothers Inc. has bought into object technology and is building software that simulates its business. It has Account objects, representing customers, Contract objects to manage agreements between parties, and Security objects that describe the properties of stocks, bonds, or options. An Account can enter into a Contract to buy a Security—just as in life. "It's more toward the reality of what's actually happening," observes Shearson Vice-President Frank Filippis.

Once objects have been built and tested, it's fairly simple to clone them for new products or services. Now, when Lehman wants to sell a new type of security, the programmers just tell the computer the special attributes of this new instrument. The Security object then automatically gives birth to a program that inherits all its generalized traits, plus the unique new attributes. "We can model all types of securities this way," says Filippis.

The upshot is a system that can keep up with business changes. At Unum Life Insurance Co., for example, whenever a state regulation changed in the past, programmers for the Portland (Me.)-based insurer had to scramble. But now, using objects, they can do such updates in one-third the time—and create software that's far more usable by nontechies, says Barby Muller, a technology manager. In some cases, "instead of the programmers, the business people can make changes to the software," she says.

Another big benefit: By building programs from prefab objects, you avoid the kind of "spaghetti" code that programmers commonly use to patch new functions onto old systems. These little programs can make software maintenance—usually the biggest cost in running a computer center—a nightmare. Brooklyn Union Gas Co. recently scrapped a 13-year-old customer information system on its mainframe that had become so huge and inflexible that the company couldn't respond to the needs of its 1 million customers. With Andersen Consulting, it created an object-based program that's 40% smaller yet does more. And the company expects it to last 20 years—on a fraction of the old maintenance budget.

Such success stories are attracting more converts. In a recent survey by researcher International Data Corp., 70% of large U. S. corporations said they are programming with objects or plan to do so soon. The main motivator? Money. Shearson's Filippis claims that his group has cut 30% from development costs. He reckons the company could save millions more if every department shared a central object library.

Hard to imagine that a single technical advance can do all this—drastically improve programmer productivity, create more reliable software, and give computers a childlike simplicity? Surely, there must be a catch. There are several.

Among the most formidable: It takes a lot of careful planning to create objects. Software designers not only have to figure out what each building block should do but they also must anticipate how each will work with thousands of other objects. "It takes a lot of engineering to make things look simple and easy," warns Stroustrup of Bell Labs.

PROJECT PINK. An even bigger obstacle may be standards. The big payoff from object-oriented software will come when there are common ways to shuttle objects between different computers. To that end, more than 160 computer and software makers and customers have joined the Object Management Group. Its goal is to create an electronic system to distribute software objects, such as multimedia documents, across a network, regardless of the type of computers that are on it. Digital Equipment, Sun, and Hewlett-Packard are now col-

BUSINESSPEOPLE WANT TO 'CREATE A FULL SIMULATION'
ADELE GOLDBERG PARC-PLACE SYSTEMS

TOP TO BOTTOM PHOTOGRAPHS BY JOHN MADERE; JOHN HARDING

laborating to produce the software.

Meanwhile, the Apple-IBM camp is working furiously to create a standards-setting object-oriented operating system—the basic program that runs a computer. That effort, say industry-watchers, will be based on Pink, an object-oriented system under development at Apple. It also will include technology that IBM acquired with the purchase of Metaphor Computer Systems, a software company headed by David E. Liddle, another Xerox PARC alumnus.

Companies that will compete with the IBM-Apple alliance—Sun Microsystems, Microsoft, and NeXT—argue that it's not necessary to build an all-new operating system to deliver the benefits of object-based software. "That's not a very realistic scenario," says Gates, who plans to slowly add object-based technology to Microsoft's operating systems.

'BLOATWARE.' Liddle contends that without an object-oriented operating system, customers won't realize the efficiencies inherent in the new technology. Worse, they'll be stuck with poor applications programs—what he calls "bloatware." These are aging packages to which hundreds of features have been added to make them "new" and "improved"—but almost impossible to master. A better idea, he says, would be to make nifty new features freestanding objects, easily accessed by any program. Such common objects may even be included with the operating system that IBM and Apple are building.

Eventually, a whole new way of selling software may emerge. In a market of interchangeable, plug-and-play objects, you might shop for pieces separately and compile your own custom software. Chunks of programs may be sold like hardware components. "You can walk into a Radio Shack and buy a chip or circuit that does a specific function," says Chuck Duff, founder of Whitewater Group, which makes programs to write object-oriented software. "That needs to happen for software."

How quickly object technology will sweep the industry is anybody's guess. Certainly, it's catching on with software makers and big corporations, who hope to make programming simpler and cheaper. But will it fundamentally alter the computer business, as some observers predict? Maybe. "The entire software environment needs a face-lift," notes Edward J. Zander, president of Sun's SunSoft subsidiary. Object technology by itself may not be the cure to slow growth, but it looks like a good bet for painting a happier face on the computer industry.

By John W. Verity and Evan I. Schwartz in New York, with bureau reports

ARE TODAY'S TINY STARTUPS TOMORROW'S SOFTWARE TITANS?

Not so long ago, a technology called the relational data base was languishing in deep academic obscurity. And Oracle Systems Corp., a West Coast startup among the first to sell such software, was a computer-industry nobody. Then, almost overnight, in the early 1980s, relational was hot. And by last year, Oracle had grown to be one of only three software companies to reach $1 billion in annual sales.

It's a success saga well known to a half-dozen gangly software upstarts that are aiming to outdo Oracle. Their "object bases," data-base programs built on object-oriented concepts, may be a giant step up from relational technology, and the latest stage in getting machines to store and retrieve information efficiently. The products need work, and they won't kill off relational data bases any time soon. But venture capitalists have been pumping millions into the technology. And customers are beginning to take it seriously.

RELATIONSHIPS. The startups are counting on delivering superior performance for highly complex jobs. The first "flat file" data-base programs, in the 1960s, placed data in long streams of numbers and letters. You could easily look up one bank customer's balance, but it could take many searches to locate all customers with balances over $10,000. Relational data bases organize data in rows and columns, like statistics on a baseball card. By reading down rows or across columns, the computer quickly spots useful relationships—such as how many players who hit more than 30 home runs also stole at least 30 bases.

But object bases take data retrieval even further, storing complex information that won't fit into columns and rows. So a computer can catalog things such as 3-D images, sound recordings, and photographs. Moreover, nontechies can use object bases without learning arcane software code. British Aerospace Ltd. uses an object base from Ontos Inc. in Burlington, Mass., to model the electrical wiring system for a military aircraft. For each of 20,000 wires, the object base keeps a schematic drawing, bill of materials, manufacturing information, and other data. Earlier, this was stored in separate data bases, making it difficult for engineers to get a complete picture—and multiplying the cost of managing the data.

For most businesses, however, object bases remain in the realm of techno-exotica. Altogether, object-base companies will have revenues of perhaps $15 million this year. And today's buyers are mostly testing the technology in small pilot projects. "I'm constantly amazed how little [computer buyers] know about this technology," says John W. Jarve, a partner in Menlo Ventures, which has pumped $2 million into startup Objectivity Inc.

The upside: As businesses adopt object-oriented programming and plunge into multimedia computing, they will need to manage swelling libraries of reusable objects, voice recordings, and videos. So, says market researcher International Data Corp., object-base sales should hit $446 million by 1996.

That's not a big enough chunk of the multibillion-dollar data-base software market to worry leaders such as Oracle and IBM—or to make them rush into object bases. As their customers begin using object-based technologies, Oracle plans simply to add some object storage and retrieval techniques to its software, says Robert N. Miner, co-founder and senior vice-president. "I was nervous that we were going to be blindsided," he says. "But now I think that we'll be able to do everything they do before they do everything we do." If he's wrong, there may be a new Oracle in the making.

By Keith H. Hammonds in Boston

THE NEW DATA-BASE RACE

INNOVATIVE SYSTEMS Specializes in investment banking applications *Founded: 1981*

OBJECT DESIGN Targeting electronic design. Marketing deal with Computervision *Founded: 1988*

OBJECTIVITY Marketing pact with Digital Equipment *Founded: 1988*

ONTOS Customers are AT&T, General Dynamics *Founded: 1985*

SERVIO Owned by Indonesian investors. Marketing pact with IBM *Founded: 1982*

VERSANT Pursuing manufacturing control and design jobs *Founded: 1988*

DATA: COMPANY REPORTS, OFFICE COMPUTING GROUP