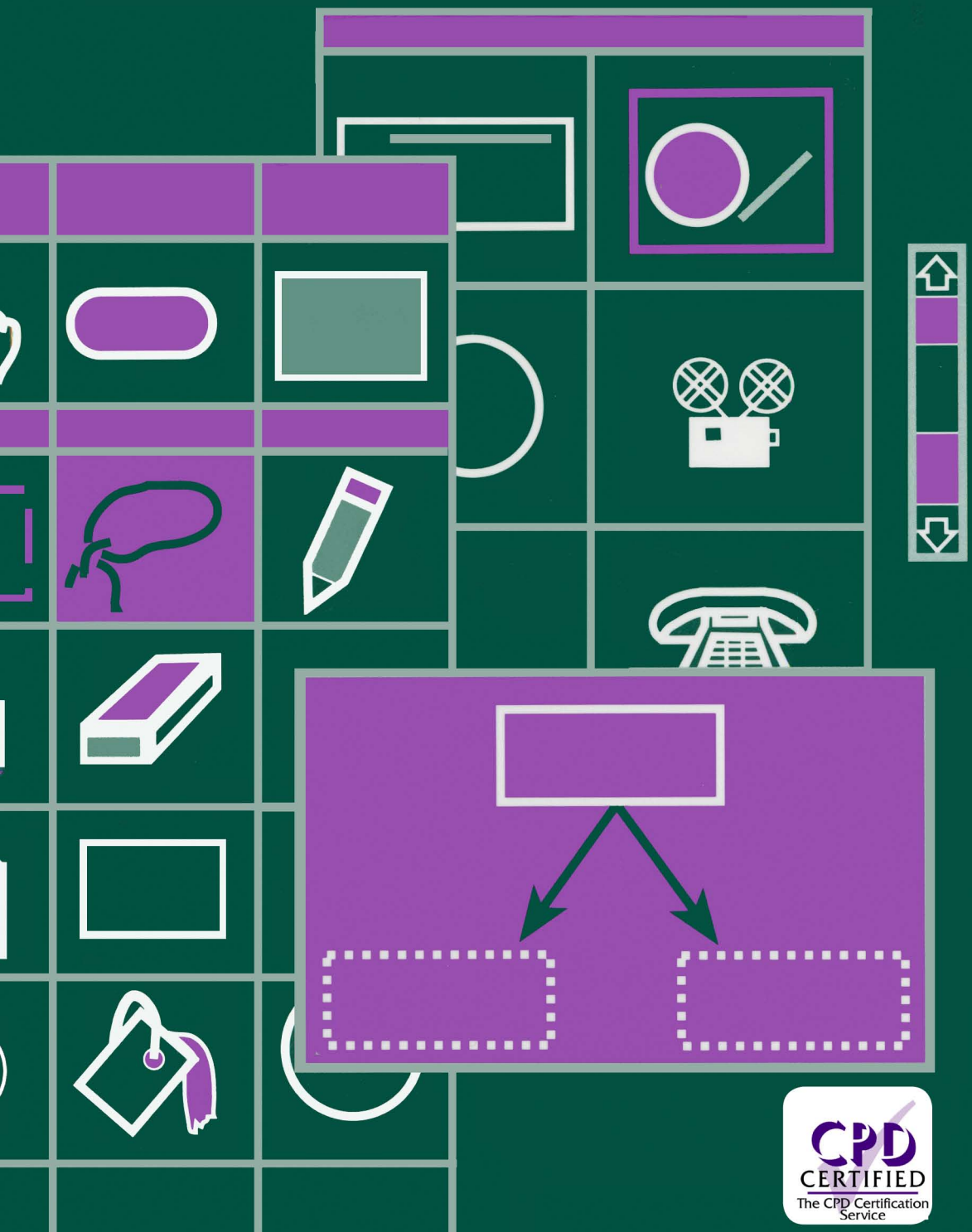
Edited by
Brad A. Myers

Languages for Developing User Interfaces



Languages for Developing User Interfaces

Trademarks

Languages for Developing User Interfaces

Edited by

Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania

with the assistance of

Mark Guzdial, *University of Michigan*
Ralph D. Hill, *Bellcore*
Bruce Horn, *Carnegie Mellon University*
Scott Hudson, *University of Arizona*
David S. Kosbie, *Carnegie Mellon University*
Gurminder Singh, *National University of Singapore*
Brad Vander Zanden, *University of Tennessee*

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Part 2: Programming Languages for Programmers

General Goals

Models for Objects and Interaction

Constraints

Concurrency and Time

Preface

Computing is evolving from batch-based applications to interactive, graphical applications. However, most user interface software is still written using languages designed for writing text-based or even batch applications, such as Fortran, Pascal, C, or Ada. Researchers are investigating new approaches that may allow the next generation of computer programming languages to better support the creation of user interface software.

In addition, user interface designers are increasingly realizing that it is important to provide a high degree of end-user customization. In many cases, it would be ideal to allow end users to create their own applications. In a sense, this is what spreadsheets allow, since they can be "programmed" by their users. The success of spreadsheets shows that end users *can* learn to program, and that environments that support end-user programming can be successful.

At the SIGCHI conference in New Orleans in May, 1991, twenty leaders of the field got together in a workshop to discuss the future of languages for programming user interface software, and for end-user programming. These twenty were chosen from over 60 people who applied. The goal of the workshop was to discuss what types of computer languages would be appropriate in the future, and begin collaborations on creating these languages. This book contains the results of those discussions.

First, Chapter 1 presents an overview of the topic, and a summary of previous work. The first day of the workshop was spent with talks from the attendees. Chapters 2 through 18 contain the written papers that accompanied their talks. During the second day of the workshop, we broke into three groups to discuss various issues in depth. Chapters 19 through 21 report on the group results. Naturally, we discovered more issues than we resolved, and Chapter 22 contains a summary of the issues that were raised. We hope this will be seen as a challenge to future language designers.

In the user interface community, this book should be of interest to creators of toolkits, UIMSs and other user interface tools, as well as people creating end-user applications that want to provide end-user customization. In the programming language community, language designers would find this book useful, since future programmers will need to write modern user interfaces with their languages.

Acknowledgements

First, we would like to thank the SIGCHI'91 conference for sponsoring this workshop, and Wayne Gray, the SIGCHI'91 Workshop Chair, for helping to organize it.

All the attendees wish to thank their organizations for supporting their attendance at the workshop.

Thanks very much to Bernita Myers and David Kosbie, who worked very hard to format this document, and convert from many different formatters into LaTeX.

Workshop Participants

In alphabetical order:

Alan Borning, University of Washington
Jeffrey L. Brandenburg, Virginia Tech
James R. Cordy, Queens University at Kingston
Michael Dertouzos, Massachusetts Institute of Technology
T.C. Nicholas Graham, GMD Karlsruhe
Mark Green, University of Alberta
Mark Guzdial, University of Michigan
H. Rex Hartson, Virginia Tech
Ralph D. Hill, Bellcore
Bruce Horn, Carnegie Mellon University
Scott Hudson, Georgia Tech
Erica Liebman, Georgia Tech
Toshiyuki Masui, SHARP Corporation
Brad A. Myers, Carnegie Mellon University
John H. Reppy, Cornell University
Bob Scheifler, Massachusetts Institute of Technology
Gurminder Singh, National University of Singapore
David Canfield Smith, Apple Computer, Inc.
Randall B. Smith, Sun Microsystems Laboratories, Inc.
Brad Vander Zanden, University of Tennessee

Organizer:

Brad A. Myers

Program Committee:

Brad A. Myers
Scott Hudson
Bruce Horn

Contributors

Alan Borning
(Chapter 11)
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, WA 98195

Robert Boyle
(Chapter 4)
University of Michigan
School of Education
610 E. University
Ann Arbor, MI 48109

Jeffrey L. Brandenburg
(Chapter 17)
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Bay-Wei Chang
(Chapter 5)
Sun Microsystems Laboratories, Inc.
MS MTV29-116
2550 Garcia Avenue
Mountain View, CA 94043-1100

James R. Cordy
(Chapters 6, 18, 21)
Department of Computing and
Information Science
Goodwin Hall
Queen University
Kingston, Ont. K7L 3N6 Canada

Michael Dertouzos
(Chapter 2)
Director, MIT Lab for
Computer Science
545 Technology Square, Room 105
Cambridge, MA 02139

Bjorn N. Freeman-Benson
(Chapter 11)
University of Victoria
Department of Computer Science
Box 3055
Victoria, B.C. V8W 3P6 Canada

Emden R. Gansner
(Chapter 14)
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

T.C. Nicholas Graham
(Chapters 16, 22)
GMD
Vincenz-Priessnitz-Str. 1
D-7500 Karlsruhe 1
Germany

Mark Guzdial
(Chapters 4, 20)
University of Michigan
Dept. of EE and CS
1101 Beal Ave.
Ann Arbor, MI 48109

H. Rex Hartson
(Chapter 17)
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Ralph D. Hill
(Chapters 9, 21)
Bellcore
445 South Street, Rm. 2D 295
Morristown, NJ 07962-1910

Deborah Hix
(Chapter 17)
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Bruce Horn
(Chapters 13, 19)
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Scott Hudson
(Chapter 7)
College of Computing
801 Atlantic Dr.
Georgia Institute of Technology
Atlanta, GA 30332-0280

Toshiyuki Masui
(Chapter 15)
Information System R&D Center
SHARP Corporation
2613-1 Ichinomoto-cho
Tenri, Nara 632, Japan

Brad A. Myers
(Chapters 1, 10, 19)
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

John H. Reppy
(Chapters 14, 20)
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Gurminder Singh
(Chapters 8, 21)
Institute of Systems Science
National University of Singapore
Kent Ridge, Singapore, 0511

David Canfield Smith
(Chapters 3, 19)
Advanced Technology Group
Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, CA 95014

Randall B. Smith
(Chapters 5, 20)
Sun Microsystems Laboratories, Inc.
MS MTV29-116
2550 Garcia Ave.
Mountain View, CA 94043-1100

Elliot Soloway
(Chapter 4)
University of Michigan
Dept. of EE and CS
1101 Beal Ave.
Ann Arbor, MI 48109

Joshua Susser
(Chapter 3)
Advanced Technology Group
Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, CA 95014

David Ungar
(Chapter 5)
Sun Microsystems Laboratories, Inc.
MS MTV29-116
2550 Garcia Ave.
Mountain View, CA 94043-1100

Brad Vander Zanden
(Chapters 12, 21)
107 Ayres Hall
Computer Science Department
University of Tennessee
Knoxville, TN 37996-1301

Peri Weingrad
(Chapter 4)
University of Michigan
Dept. of EE and CS
1101 Beal Ave.
Ann Arbor, MI 48109

Chapter 1

Introduction

Brad A. Myers

In his keynote address to the SIGCHI'90 conference, Michael Dertouzos said:

> When computers first appeared, input/output commands were a minor afterthought to cohesive, often well crafted and occasionally pretentious programming languages. Today, these commands occupy over 70 percent of a programming system's instructions. Yet they, along with the user interface structures that they define, are far from cohesive, and, at least up until now, immune to standardization. We must therefore turn our thinking around and create a new breed of programming languages that are first and foremost input/output oriented and that integrate traditional processing commands into new user-oriented structures. And just as we know today that traditional commands fall into a handful of fixed categories—decision, repetition, naming, procedure definition and use—we need to search for and identify the corresponding natural classes of commands for user interfaces. [Dertouzos 90]

Researchers in the areas of user interface software have been investigating the use of special-purpose languages for programming user interfaces for many years. For example, TIGER, in 1982, was the first system that was called a "user interface management system," and it used a special

language called TICCL to define the user interface [Kasik 82]. Many other systems in the 1980s used BNF grammars or state transition diagrams to define the user interface (see Section 1.3). Today, researchers are concentrating on new forms of object-oriented languages and features to add to them. However, no one believes that the problem is even close to being "solved."

In addition, this research has not had much effect on the computer languages being designed by researchers who call "programming languages" their primary area of interest. I recently attended a presentation about a new programming language being designed. In the section labeled "input/output" were the conventional scanf/printf (readln/writeln in Pascal) statements. When asked if he thought these were sufficient for a modern language, the presenter replied "no," but he did not know enough about the area to do better. Some people claim that programming languages should not contain *any* I/O primitives, but rather leave it to separate packages. However, this book will show that user interface programming requires a number of important features not found in most of today's languages which cannot be relegated to external packages.

Another problem is that for applications to reach their full potential, end users will have to be able to customize and even program them themselves. Today, end-user applications are getting more and more complicated, as each release adds new features. For example, version 4.0 of the Microsoft Word text editor for the Macintosh has over 280 commands. However, users often find that what they really want is a few features from one program coupled with a few from another. If an end-user programming facility was provided that allowed the users to combine these features to create their own systems, this might solve the problem. The success of spreadsheets, which allow users to create their own programs by writing formulas and macros, shows that end users can program when given the appropriate tools, and that a product based on end-user programming can succeed.

At the SIGCHI conference in 1991 in New Orleans, a workshop was held to try to bring together user interface software specialists and programming language designers, to discuss how computer languages of the future can better support the construction of applications with modern, highly-interactive user interfaces. Twenty people met for two days to dis-

cuss this topic, and this book is a result of that discussion. The rest of the introduction motivates the problem and surveys some previous approaches.

1.1 Creating User Interface Software

It is well known that programming user interfaces is difficult. Studies consistently show that the user interface portion comprises about 50% of the code and development time [Myers 92b]. There are a number of reasons that software for modern user interfaces is inherently more difficult to write than other kinds of software:

Iterative design. Because user interfaces are difficult to *design*, the initial attempts are usually not good enough, and the interface must be re-implemented [Gould 85]. This *iterative design* requires that the user interface software be repeatedly and frequently modified. As reported by Sheil [Sheil 83], "complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so." The code must therefore be written so that the user interface portion can be easily changed, preferably without affecting the other parts of the software. However, most programmers find that making this separation is difficult.

Difficult to get the screen to look attractive. It is usually difficult to use the supplied graphics packages and libraries. As a result, achieving the desired graphical appearance can be a challenge. Techniques are required to support interactive specifications of the static and dynamic appearance and behavior of the interface.

Asynchronous inputs. Direct manipulation interfaces have the characteristic that the user is in control of the interface, and can perform input at almost any time. The program must therefore be able to accept input at any time. Also, the software must usually be organized with a central *event dispatcher* loop, which accepts the input events from the user, and uses the type of the event to decide which command to execute. This is quite a different software structure than for conventional programs.

Multiple processing. Since the program must be able to accept input events at any time, but some application procedures may take a noticeable amount of time, the software is typically organized as multiple processes, so longer actions can be executed in the background. Also, the window manager will often be in a different process than the user interface software, and may send requests to the application to redraw the windows (if they become uncovered). Dealing with multiple processes means that the programmer must deal with synchronization, race conditions, and many other problems.

Efficiency. All code that interfaces to the user must operate without a noticeable delay. For example, if an object is being dragged with the mouse, it should be redrawn at least 30 times a second. This means that the programmer must often deal with all the problems of real-time programming.

Error handling. When an error happens in a user interface, it is not acceptable for the program to "crash." An appropriate message must be shown to the user, and the system must be able to recover and continue processing. This puts tremendous emphasis on robustness in the programs.

Aborts, Undo, and Help. Most interfaces should allow the user to abort an operation at any time, or ask for help. This means that the software must be organized so that the appropriate information is available so the state can be restored to before the current or previous command was started, or to tell the user what is happening.

1.2 The Problem

This book covers two different kinds of programming: allowing end-users who do not have any formal training in programming to extensively customize their interfaces, and conventional implementation of user interfaces.

1.2.1 End User Programming

Users of spreadsheets and database packages write programs in the specialized languages of those systems. A large number of people have mastered the skills needed to write these programs, and it has been argued that the programmability of these tools is the primary key to their success: the user can get them to do what he or she wants. However, most other applications on computers are not programmable, and there is certainly no uniform language that can be used across different applications. Therefore, a challenge for the future is to develop a mechanism that allows end users to customize all applications.

We classify this as a style of programming because users will need program-like capabilities, such as conditionals, loops, and variables. For example, the user in a "visual shell" or desktop, like the Macintosh Finder, might want to say "delete all backup copies of files older than January 1988 if the associated original files are on the disk." This clearly requires a loop over all files, variables to hold the backup file and the associated original file, and a conditional to test the age. Since reliable natural language understanding is a long way off, we need some other way for the end user to express this request. However, there is plenty of evidence that end users find conventional programming difficult if not impossible [Shneiderman 80]. How can end users specify complex requests? We feel that programmability will be an important component of future user interfaces.

1.2.2 Conventional Programming Languages

We have identified two important classes of programmers who need to create user interfaces for programs: novice programmers and professional programmers. Neither has adequate tools today.

Novice Programmers

Students who are learning to program today have used video games and computers such as the Macintosh, which have sophisticated graphics and user interfaces. When they learn to program, they expect to be able to create similar systems. However, the programming languages in use today, such

as C and Pascal, have the same old I/O primitives as Fortran: read and write a string. As a result, large and complex external libraries of routines are needed to perform graphical interaction.

Current programming languages generally support simple textual input and output, and the canonical first program prints "hello world" on the screen. In most programming languages, this will be a one to three line program. For the future, however, new programmers will want to create graphical, highly-interactive programs. Therefore, our goal for a future computer language would be to make creating a blue rectangle that would follow the mouse be as easy as writing "hello world" today.

Therefore, work must be concentrated on creating the appropriate abstractions for hiding the complexities of today's window managers and graphics packages, just as languages of the present hide the complexities of how to make strings appear on the screen. What new paradigms and techniques can be used in future languages so that novice programmers can learn how to create graphical, interactive applications in the first few weeks? For example, the moving blue rectangle program should be only 5 to 10 lines.

Programming for Professional Programmers

A wide variety of tools have been created to help with implementing user interface software, including toolkits and User Interface Management Systems (UIMSs). Many of these have created their own new programming language. For example, the popular Xt toolkit for X, in which both Motif and Open Look are implemented, created its own object-oriented language embedded in C (see Section 1.3.2). The Garnet system defines its own embedded language using Common Lisp (Chapter 10). Current research in user interface tools focuses on object-oriented techniques, constraints, and parallelism, which should be built-in features of programming languages. Therefore, a discussion of future user interface tools must include a discussion of the design for the language the programmer will use. What are the goals, features, and characteristics for future languages for programming user interface software?

General Problems with Programming Languages

In summary, the problems we have identified with programming user interfaces in conventional languages include:

1. Lack of appropriate I/O mechanisms. Conventional languages still provide only limited character input and output, which supports a textual question-and-answer interaction model that is 40 years old. It is well recognized that this creates user interfaces that are modal and hard to use.

2. Lack of inexpensive multi-processing and real-time programming. Handling asynchronous input events from the user while supplying real-time feedback often requires multi-processing.

3. Ineffective object-oriented paradigms. It is the conventional wisdom that all user interface software should be programmed using object-oriented techniques. All modern user interface toolkits use this technology, but some modern languages are still not object-oriented.

4. No rapid prototyping. Many languages are designed to support the conventional software engineering model, where software is first specified, then designed, and finally implemented. However, user interface software generally requires many iterations of prototypes and re-implementation [Gould 85].

5. Inappropriate representation for programs. The textual representation of programs makes it difficult to specify graphical entities, but graphical representations to date have failed to achieve the compactness and flexibility of text.

6. Lack of various new features being investigated by user interface researchers, such as constraints, event-handlers, and incremental recomputation (these are explained in the following sections).

1.3 Survey

1.3.1 Programming Languages

Programming languages have long had embedded commands for performing input and output. However:

> Input and output are perhaps the most systematically neglected features of programming languages. They are usually ad hoc, and they are usually poorly integrated with the other facilities of their hosts—the languages in which they are embedded.... The situation was bad enough before the introduction of abstract data types and interactive graphic displays, but these additional complications have overburdened the classical ad hoc input and output mechanisms beyond their design limitations. [Shaw 86]

Fortran, developed in the mid-1950s, provided sophisticated text formatting and reading facilities, so that the programmer could control the exact format of the output and input. The roots of the model are based on batch processing of lines of text or streams of characters. Later languages have advanced little in this area, and still use similar mechanisms. For example, the facilities provided by C (1972), Pascal (1975), Common Lisp (1984), and even modern languages such as Ada (1983), Turing (1983—see Chapter 18), and the functional language Standard ML (1985) [Milner 90], only support text writing and reading, with varying levels of control over the formatting. These text I/O primitives are often built-in mechanisms because, unlike other functions in the language, they usually take a variable number of parameters. Some other modern languages, such as Mesa [Mitchell 79], do not have *any* built-in I/O mechanisms.

The built-in primitives only support the question-and-answer style of user interface, which is no longer very popular. The system prints a prompt (using something like writeln or printf) and the user is supposed to type in the answer (using readln or scanf). Notice that the program is fully in control, and the user has no option to perform a different action, ask for help, or revise earlier answers. To create graphical or direct manipulation style interfaces in any of these languages, the programmer must ignore the

built-in primitives and use a separate library of routines, which is not part of the language standard.

Some argue that it is considered a good design principal to leave I/O out of the language definition and instead define it as part of the standard libraries. The lesson of PL/I shows that trying to incorporate all of the useful and reasonable semantics of I/O in a language leads to a bad design. However, as discussed above in Section 1.2.2, even without specific I/O mechanisms, there are many other features that *are* considered appropriate to be part of the design that have a significant impact on user interface software.

1.3.2 Languages for Programming User Interface Systems

The field of software for user interfaces has been actively researched for many years, and there are a number of good surveys (e.g., [Hartson 89b, Myers 89a, Myers 92a]) and books (e.g., [Bass 91]) about the topic. There is also an annual conference devoted to user interface software, called the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST). Since the area is so broad, this section does not try to cover all the existing systems, but rather provides an overview of the various language approaches that have been used.

From the beginning, tools for creating user interface software have used special-purpose languages. When a system includes a special language for defining the user interface, it is often called a *User Interface Management System* (UIMS). Often, the assumption is that the user interface will be programmed in this special language, but the *application* (all of the code that is not the user interface) will be programmed in a conventional language. The following sections discuss some of the forms the special languages have used.

State Transition Diagrams

Since many parts of user interfaces involve handling a sequence of input events, it is natural to think of using a *state transition diagram* (essentially a finite state machine) to define the interface. A transition network consists of a set of states, with arcs out of each state labeled with the input tokens that will cause a transition to the state at the other end of the arc (see Figure

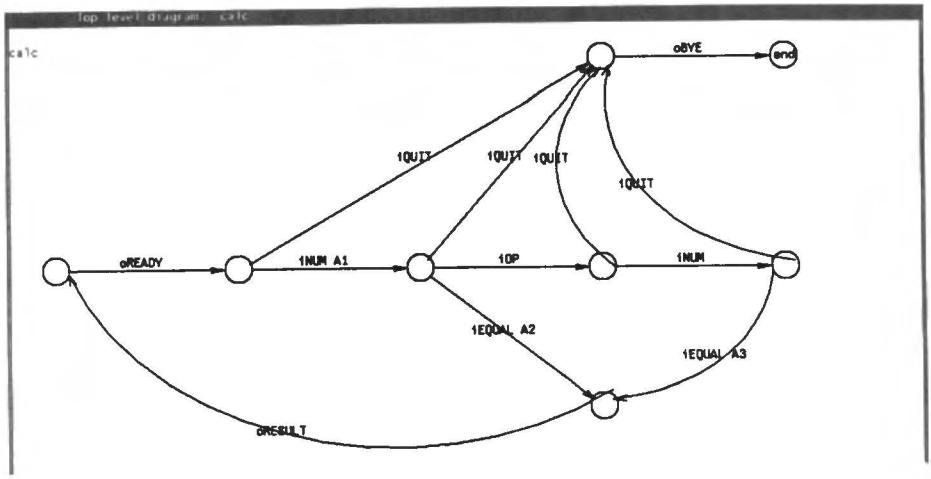


Figure 1.1: State diagram description of a simple desk calculator [Jacob 85b].

1.1). When the user performs the action on the arc, the system goes to the next state. In order to have some action happen when the transition takes place, many systems allow the programmer to also specify on arcs or states the output that will be shown to the user, and application functions to be called.

Newman used a state transition diagram in what was apparently the first UIMS [Newman 68]. Jacob added the ability to have procedural abstraction, so that the label on an arc could actually be a call to a sub-diagram [Jacob 85b]. Figure 1.1 is a view of this system.

State diagram UIMSs are most useful for creating user interfaces where a large amount of syntactic parsing is necessary or when the user interface has a large number of modes (each state is really a mode). However, most highly-interactive systems attempt to be mostly "mode-free," which means that at each point, the user has a wide variety of choices of what to do. This requires a large number of arcs out of each state, so state diagram UIMSs have not been successful for these interfaces. If the user can give parameters to a function in any order, a state transition diagram must have a different set of transitions for each order. In addition, state diagrams cannot handle interfaces where the user can operate on multiple objects at the same time

(possibly using multiple input devices concurrently). Another problem is that they tend to get very confusing for large interfaces, since they get to be a "maze of wires" and off-page (or off-screen) arcs can be hard to follow.

Recognizing these problems, but still trying to retain the perspicuity of state transition diagrams, Jacob [Jacob 86] created a new formalism, which is a combination of state diagrams with a form of event languages (see below). There can be multiple diagrams active at the same time, and flow of control transfers from one to another in a co-routine fashion. The system can create various forms of direct manipulation interfaces. However, very few state transition systems are in use today.

Because state transition diagrams are naturally graphical, most systems have allowed the user to enter them using a graphical editor. They are therefore *Visual Programming Languages* [Myers 90d]. However, some early systems required the programmer to enter the diagrams using a textual language.

Grammars

For user interfaces that use command languages or other text-based input, it seems natural to use a context-free grammar to parse the input. Therefore, some early UIMSs allowed the programmer to define the syntax of the expected input using a BNF grammar. Tools such as YACC and LEX under Unix can then be used to generate a parser automatically. The Syngraph (SYNtax directed GRAPHics) UIMS [Olsen 83] is a system that tried to extend this idea to graphical programs, by having the syntax of the interface defined in an extended BNF. However, all of the problems mentioned above for state transition diagrams also apply to grammars. In addition, programmers usually find it very difficult to visualize the resulting sentences from a grammar. Consequently, grammars are usually only used for describing highly-constrained, textual input.

Event Languages and Production Systems

When the user hits a keyboard key or a mouse button, window systems create an "event" structure containing various pieces of information about the input event. This structure is put in a queue, and the user interface software must take the events out of the queue and process them. Therefore,

it seems natural to create a system that is organized around event handlers. Each handler is a small piece of code that is called by the system when the appropriate event occurs. Usually the event can be qualified by other conditions, for example, "left mouse button down while inside the 'Reset' button." The handler might perform some output, call an application procedure, or generate a synthetic event to cause other handlers to operate. It has been shown that event systems are more flexible than either state transition diagrams or grammars [Green 86].

The ALGAE system [Flecchia 87] uses an event language which is an extension of Pascal. The user interface is programmed as a set of small event handlers, which ALGAE compiles into conventional code. Sassafras, which implements an Event Response System (ERS) [Hill 86], uses a similar idea, but with an entirely different syntax. This system also adds local variables called "flags" to help specify the flow of control.

The HyperTalk language used to program in Apple's HyperCard is a recent example of an event language. The user writes code that is invoked when a button is hit or other event occurs. HyperTalk is further discussed in Section 1.3.3.

Event systems are much like production systems used by some AI (artificial intelligence) systems. In production systems, there are many "rules" of the form **if** `test` **then** `action`. The system repeatedly tries to find a rule whose test passes, and then executes its action. The PPS system [Olsen 90b] uses a production system approach, which is more general than an event system.

One nice thing about event languages is that they can easily handle multiple processes, which can be important in user interfaces. One of the problems with event languages is that it is often very difficult to create correct code, since the flow of control is not localized and small changes in one part can affect many different pieces of the program. It is also often difficult for the designer to understand the code once it reaches a non-trivial size. Hill [Hill 86] claims that these problems can be solved if the event language provides appropriate modularization mechanisms.

Declarative Languages

Another approach is to try to define a language that is declarative (stating what should happen) rather than procedural (how to make it hap-

pen). Cousin [Hayes 85] and the commercial product Open Dialogue from Apollo Computer, Inc. (now part of Hewlett-Packard) both allow the designer to specify user interfaces in this manner. The user interfaces supported are basically forms, where fields can be text which is typed by the user, or options selected using menus or buttons. There are also graphic output areas that the application can use in whatever manner desired. The application program is connected to the user interface through "variables," which both can set and access.

The advantage of using declarative languages is that the user interface designer does not have to worry about the time sequence of events, and can concentrate on the information that needs to be passed back and forth. The disadvantage is that only certain types of interfaces can be provided this way, and the rest must be programmed by hand in the "graphic areas" provided to application programs. The kinds of interactions available are preprogrammed and fixed. In particular, these systems provide no support for such things as dragging graphical objects, rubber-band lines, or drawing new graphical objects.

Constraint Languages

"Constraints" are relationships that are declared once and maintained automatically by the system. They are often considered declarative languages, since the programmer does not specify how to solve the constraints, only what the relationships should be. However, we have not included constraint languages in the previous section because they have a quite different form and use than the systems described above. Unlike Cousin and Open Dialogue, constraint languages are most often used for the *dynamic* parts of an application. For example, the programmer might declare that a line should stay attached to a box. Then, when the user moves the box, the system will automatically move the line also.

Constraint languages have been widely used to design user interfaces in research systems [Borning 86], and Chapters 11 through 13 discuss some modern constraint systems in more detail. Early constraint systems include Sketchpad [Sutherland 63a, Sutherland 63b] which pioneered the use of graphical constraints in a drawing editor in the early 1960s, and ThingLab [Borning 79, Borning 81] which used constraints for graphical simulation. More recently ThingLab has been refined to aid in the generation of user

interfaces [Freeman-Benson 90c]. GROW [Barth 86] was perhaps the first user interface development system that employed constraints.

The advantage of constraint languages is that it is convenient for the programmer not to have to keep track of all the relationships and how to maintain them when changes happen. A disadvantage is that today's constraint solvers are usually inefficient in space and time. In addition, a complex network of constraints can be difficult to debug, since changing a value can have non-local effects if constraints depend on it.

High-Level Specification Languages

Some research systems are investigating allowing the programmer to define a high-level specification of the application functionality, and automatically generating a user interface from that. For example, in IDL [Foley 88], the programmer gives the application procedures along with pre- and post-conditions for each. From these, the system can create a preliminary interactive user interface, which the programmer can then modify to be more attractive and easier to use. Mickey [Olsen 89] uses a Pascal definition of the application procedures to be called and variables to be set along with special comments, to generate a Macintosh menu and dialog-box interface.

The advantage of using high-level specification languages is that the programmer does not need to worry much about the user interface, and can concentrate on the application functionality. The disadvantages are that the systems rarely create good user interfaces, so tinkering is necessary, and the systems are limited in the forms of interfaces they can create.

Object–Oriented Languages

Many user interface development systems are based on existing object-oriented languages. For example, InterViews [Linton 89] uses C++, and GWUIMS [Sibert 88] uses the Flavors object system in Lisp. In fact, one of the chief motivations for Smalltalk, the first successful object system, was that it would be easier to create user interface software.

In addition, special object-oriented languages have been created specifically to support user interface development. These include Object Pascal, which was created by Apple as part of the MacApp program development system [Wilson 90], and the Garnet Object System (see Chapter 10).

Also, some toolkits, such as Xt [McCormack 88] and Andrew [Palay 88], have invented their own object systems. In these two cases, the underlying language is C, and the tool developers felt that other object-oriented languages, such as C++, were inadequate, so they developed their own object-oriented systems using extensions to C.

Object-based systems typically provide the higher-level "classes" that handle the default behavior and the user interface designer provides specializations of these classes to deal with specific behavior desired in the user interface. This uses the inheritance mechanism built into object-oriented languages.

The advantages of using an object-oriented approach are well-known. The entities on the screen are naturally modeled by objects receiving messages, since they need to respond to events. In addition, the inheritance mechanism of object systems makes it easier to reuse code since standard mechanisms can be defined, and the programmer can override only those that are specific to the particular application. Virtually all modern user interface software environments are object-oriented.

1.3.3 Languages for End-User Programming

In the old days, computers were mostly used by programmers or scientists who knew how to program them using conventional programming languages. Today, however, the vast majority of computer users do not have any training in computer programming. However, these users find that they still need many of the capabilities that programming provides: the ability to direct the computer to perform a specific user-defined task, and to customize existing applications. Many approaches have been tried to provide this capability to users.

Clearly, the most successful end-user programming systems are spreadsheets, such as Lotus 1-2-3. Spreadsheets are enormously popular for personal-computer users, and some claim that spreadsheets are the primary reason most people buy personal computers. Spreadsheet users write programs by entering formulas into cells, and by creating macros of spreadsheet operations. Why spreadsheets have been easy to use and program has been studied by many researchers [Kay 84, Hutchins 86, Lewis 87, Nardi 90] (see also Chapter 19).

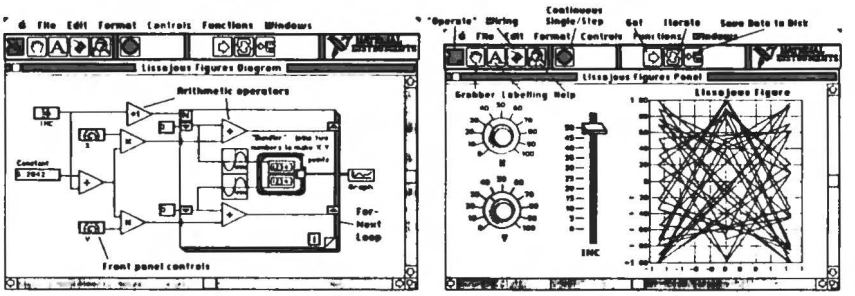
Another popular product for personal computers is database programs. These systems, such as DBASE, allow the end user to create database query programs to find information stored in the database.

The HyperCard program from Apple for the Macintosh allows end users to create applications. It is primarily good for making "forms" (called cards) containing fill-in fields and buttons. The buttons can transfer to other cards or perform other actions. If the user wants a complex action to happen, this can be programmed using the HyperTalk scripting language. However, most people who do not understand how to program have great difficulty writing HyperTalk programs.

Creating programs using graphics has long been touted as a method for making programming easy enough for end users. Many "Visual Programming Languages" [Myers 90d] have been designed to provide programming capabilities to non-programmers. For example, the LabView product for the Macintosh allows scientists to create dataflow diagrams to create a control panel for external instruments [Labview 89]. The processing of the data and control signals can be defined using icons connected by graphical wires (see Figure 1.2). Another example is Authorware, which uses a flowchart style graphical language to allow schoolteachers to design educational software [Authorware 91]. In Chapter 6, Cordy describes a new visual language, based on a functional model, rather than the imperative model used by most visual languages.

The advantages of graphical approaches are that there is usually no syntax to learn, so it is easier to create the programs, and often the two-dimensional presentation can help users understand the flow of control. In general, however, graphical programming has not been a panacea for end users. The *concepts* of programming, such as conditionals, iterations, and variables, are often hard for people to understand, and the graphical languages do not hide these. Also, graphical programs can be hard to read when they get larger than a few operations, since often the programs take up much more space than a textual program, and some forms can become a "maze of wires."

Spreadsheet systems, such as Lotus 1-2-3 and Microsoft Excel, have long allowed users to create "macros," which are a recording of a sequence of operations that can be replayed later. Research systems have investigated sophisticated macro recorders for Visual Shells [Halbert 84]. Commercial



(a) (b)

Figure 1.2: A LabVIEW window (a) in which a program to generate a graph has been entered. The resulting user interface after the program has been hidden is shown in (b).

macro recorders also exist for mouse-based operating systems like the Macintosh. In many systems, the recording can be edited, and control structures such as conditionals and iterations can be added, which converts the macros into full-fledged programming systems. Sometimes the macro is recorded as a text file, and then edited directly. Other times, for example in Tempo II Plus [Tempo2 91], a series of dialog boxes is used to guide the user's editing.

The advantage of macro scripting is that the user can just operate the system normally and the commands will be remembered. The disadvantages are that this technique cannot be used to create new applications (only to more effectively give commands to existing ones), and it is difficult for users to specify control structures and variables in most macro languages.

1.4 Summary

In general, the existing approaches to user interface programming, either for end users or professional programmers, have proven to be quite difficult to use. Further research is clearly needed to find better paradigms and ways to present important features. The rest of this book discusses some current and future research on this problem.

Part 1

Programming Languages for End Users

Chapter 2

The User Interface is *The* Language

Michael L. Dertouzos

The 1970 programming manual for Dartmouth Basic describes an arsenal of some 80 instructions, 10% of which are dedicated to Input/Output (I/O). Twenty years later, the 1990 Microsoft Basic manual for the Macintosh describes some 400 instructions (including relevant toolbox calls), 70% of which deal with I/O. Figure 2.1 illustrates this difference and shows how the I/O instructions are distributed among their various categories.

Despite the obvious shift in demand, reflected by this evolutionary change, and notwithstanding current rhetoric about new software environments, little has changed in the fundamental structure of programming languages since Fortran. The step from machine language to Fortran has yet to be dwarfed by a step from Fortran to anything else! Contemporary languages still carry the same basic classes of commands for decision, repetition, binding and unbinding, arithmetic and math, procedure definition and use, as well as the separable and increasingly bulkier input output (I/O) or user interface commands. It should not be too surprising that this structural inertia is accompanied by a corresponding functional feebleness—programming productivity has barely budged beyond about 1% per year, by even the most optimistic of counts, and programming continues to be out of the reach of most people.

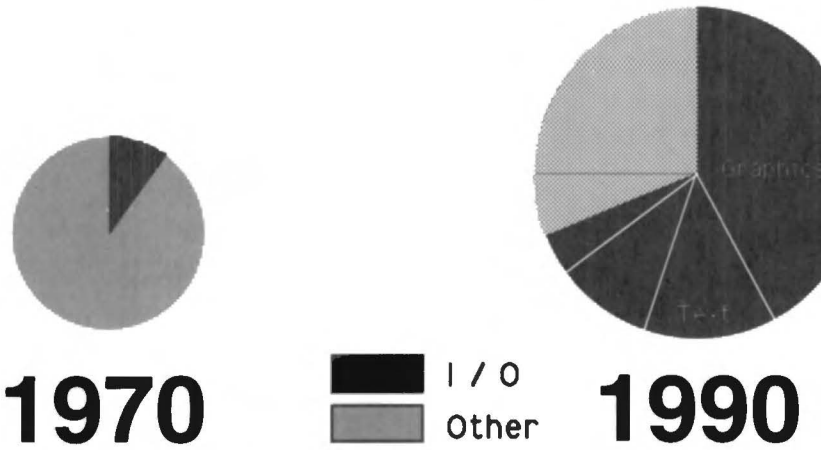


Figure 2.1: Input/output instructions - BASIC programming language.

These observations led me, in the ACM's Conference on Human Factors in Computing Systems (SIGCHI'90) [Dertouzos 90] keynote address, to call for the creation of a new breed of programming languages that would make programming much easier, much more accessible, and much more fun than it is today, blurring the distinctions we now make between programmers and users, processors and peripherals, languages and operating systems. In my view, this can happen only if the programming language becomes **rooted in** and fully integrated with the user interface—a reversal of our traditional thinking and indeed of the title of this book which presupposes a distinction between language and user interface.

The remainder of this chapter discusses a few key characteristics that such a language should have:

2.1 Out–In Programming Process

An essential ingredient of this new vision is that the programming process would start with the construction of the user interface. After all, doing something purposeful by and for the user is the entire purpose of the program that is about to be born. This means that the new language should have tools that can easily create buttons, menus, dialog boxes, windows, pictures, and sounds for input and output as well as other artifacts close to

the user that are deemed natural and purposeful for the task at hand. This is clearly a creative activity with a great deal of potential and not too great a learning cost, since all the user does is select and arrange familiar gadgets like windows, buttons, and menus.

So far, I have described what in today's vernacular would be called a user interface prototyping language (e.g., *Prototyper* by Smethers Barnes for the Macintosh). Unfortunately such prototyping software stops being useful exactly at the most interesting point of the programming process: Once the interface is designed, reams of code are generated, and the user who wishes to go further must leave the familiar and personally interesting world of the interface that she has just prototyped and plunge into the antiquated, unproductive, and unbearably detailed world of conventional programming languages like Pascal, and C—a world that caters much more to what computers like rather than to what is easy and natural for people to do.

What is needed instead is the ability to proceed smoothly from prototyping the user interface to the next natural stage—namely to what should happen when each button is activated, each menu item is selected, and each sound is made or spoken. In the language of my dreams this would be done easily by "flipping" each button that has been prototyped and specifying "behind" it what action should be taken when the button is activated by the user—akin to the spreadsheet metaphor where behind each cell may lie a formula or procedure that determines the cell's contents. This means that the programming environment of this new language **should be very rich in pre-programmed entities that can be simply selected** and that can do a lot of useful things near the I/O level of human interest. In other words, a style characterized simply by inputs, and actions caused by these inputs, which we might call **shallow programming** is good and productive and should be encouraged.

More generally, this process of **out–in** programming would continue from the user interface design to progressively deeper inner structures for more complex programs. At any time in this process, the programmer would have the ability to run the program under development with the flip of a lever, and without having to stand on his head in order to use separate build, compile, and link procedures that characterize today's development systems.

A considerably greater and more intelligent amount of compilation and run-time decision making would underlie this process, proceeding incrementally and invisibly to the user, as the program is built. This process would yield a finished prototype application, without additional fanfare, at any stage of the development process, and certainly upon its termination.

2.2 Total Environment Integration

Since a successful new language should survive for a long time, it should try to anticipate future developments. We are thus necessarily led to some crystal ball gazing.

Computer technology is growing in three important directions: Locally, the silicon used in computer circuits will be increasingly organized into multiprocessor architectures, roughly for the same reason that it is easier to harness many horses rather than grow one huge horse with the same total strength. Globally, these parallel computers will be increasingly interconnected to one another, forming networks at many levels of granularity, according to the aggregation of the population they serve—a single building, a building complex, or organizations spanning cities, and even continents. Finally, people will utilize tomorrow's computers only if they can easily communicate their wishes to these machines using speech, handwriting, pictures, and text and only if they can derive real benefits from such interaction.

These three observations suggest that the designers of future languages should keep in mind that the target of their endeavors is a system like that of Figure 2.2. In words: **we should strive to create programming languages and software systems that make networked multiprocessors easy to use through interaction by normal people toward the fulfillment of tangible goals.**

Accordingly, tomorrow's languages should include **integrally, rather than as afterthoughts**: (1) input/output capabilities for multiple media, (2) communication capabilities for dealing with users and servers over networks,, and (3) capabilities for controlling multiple resources. In short, future languages should include integrated access to this broader environment, for the simple reason that these capabilities will be present and should

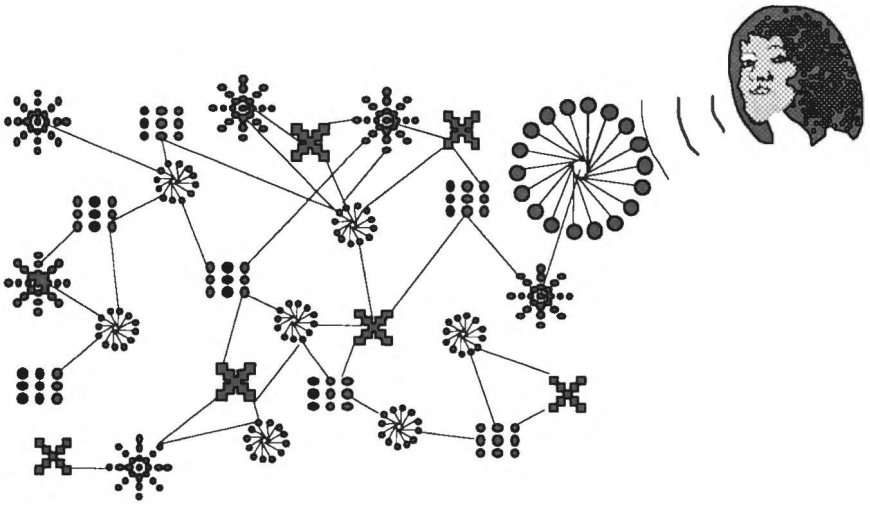


Figure 2.2: Target of new languages: networked multiprocessors that are easy to use.

be controllable by everyone. I will not discuss in this write-up the extent to which new languages should have implicit or explicit control of parallelism. The knowledge around this important issue is still accumulating. Until we know more, the new language I am after should try to achieve as much as possible implicitly, in the interest of ease of use.

2.3 Simplification

The combination of so many different kinds of information and information processing in new languages creates a big opportunity for designers to economize: Consider, for example, the many different commands programmers and users invoke today to name programming entities. Some of these are assignment statements within programming languages, file creation and renaming commands in operating systems, communication port naming commands, startup shell naming commands, naming of buttons, of sounds, of pictures; as well as the myriad of naming commands within some 10,000 packaged software applications. The opportunity to integrate all of these essentially identical activities under one generic naming com-

mand is suggestive of what this kind of simplification might accomplish in reducing the complexity of the immediate user environment.

A straightforward inspection of the commands found in today's languages, operating systems, communication systems, and applications reveals the following broad classes, under which commands might be combined and simplified:

Input–output

1. Communication with other users and programs
2. Menu selection input
3. Buttons input
4. Text input and output
5. Static picture input and output for displays, printers, and like devices
6. Window related commands
7. Sound input and output
8. Video input and output

Internal Information - to - Internal Information

9. Decision and control of computational flow
10. Navigation through pre-programmed entities
11. Math, functions, expressions
12. Move and Build (e.g., join, cons) commands, including procedures
13. Data and their organization (databases)
14. Error related, access control, and miscellaneous commands

There are obviously many other ways to categorize and simplify the millions of different commands in use today by applications and languages to control the computing environment. The important observation here is that under the current scheme, people have to learn new ways for expressing familiar commands for each application that they possess. Thus a central research question is "Under what categorization scheme can we make substantial gains in commonality, simplification and integration, hence in ease of learning and simplicity of use?" Good answers to this question, that minimize the number of different commands we need to remember, should act as a powerful guide in the design of new languages.

Commands are not the only targets for simplification. Common interfaces for data representation such as text, pictures, tables, graphs, charts, drawings, sounds, and video would go a long way toward simplifying the coupling of programs to one another.

With these concepts in mind, we can now see how traditional programming languages and operating systems would become blurred into a new kind of language that I call **My Virtual Computer (MVC)**, shown in Figure 2.3. The figure illustrates commands and data as standard interfaces that are accessible to users/programmers. In the figure these are drawn as **rails** to emphasize their role as solid interfaces. The rectangular solids on top of MVC represent a new class of "applications" that would plug into these rails and would run on this new platform. These applications would use the common MVC rails—command interfaces like **name** and **move** and common data interfaces like **text** and **video**. The boxes at the bottom of the figure represent the different machines, individually or in networks, on which MVC would run.

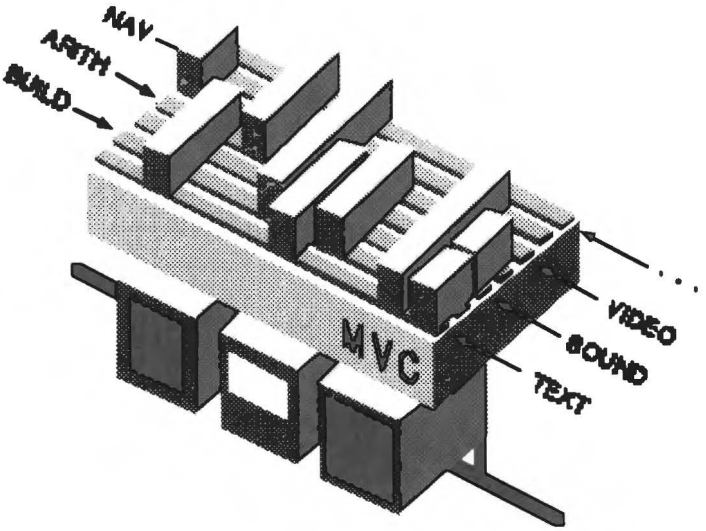


Figure 2.3: User's view of envisioned language in My Virtual Computer.

Qualitatively, the above suggestion sounds like something we already do. Quantitatively, we do it so minimally that it is essentially nonexistent: Today, each piece of application software carries along its own versions of these potentially common commands and data. I estimate that this excess baggage, whose idiosyncrasies have to be re-learned from application to application, occupies, on the average, more than 70% of each application's arsenal of command and data entities. This is a totally unacceptable

learning burden, requiring people to remember the contents of 35 manuals describing essentially the same commands in slightly different ways—if they want to use fifty applications effectively!

By contrast, MVC applications would not require as much learning, since a far larger number of common commands and interfaces would be provided by the MVC rails. Users would be the real beneficiaries of this simplification, since they would be able to learn and use new application modules far more easily than is the case today.

The introduction of commands like **file**, **open**, **save**, **cut**, and **paste** in the Apple Macintosh is a good example of common command interfaces, as are **text** and **pict** of data interfaces. This simplification has been responsible for a good deal of the success and appeal of that machine: People appreciate knowing that there is a familiar lever in a familiar place, which when pulled does familiar things. What I am advocating here is that (1) we carry this idea far beyond the Macintosh level to **all possible common** I/O and information processing commands and (2) that we plug into these rails specialized modules that are closer to user's interests as discussed next.

2.4 Extensions to Specialized Concepts—Application Modules

Suppose that I ask you to

> *write a small program that keeps track of my checkbook entries, including the category of expenditure of each check, so that the program can give me at any time a report of checks written and totals under each such category.*

Assuming that you have understood the above request and that you are willing to comply, I have in effect programmed you to develop a desired program in less than 14 seconds. The outcome of your programming effort, using spreadsheets or, more tediously, a programming language—is likely to be acceptable to me even though I did not give you too many details.

The question of interest here is: "How is it that I can successfully program you in 14 seconds and you need 100 to 1000 times more time to program the computer?" A good part of the answer must be that you and

I share a few common concepts like *checkbook*, *category of expenditure*, *report and total*, which you understand effortlessly but must painstakingly program to a concept-free machine.

Can we evolve our programming language to get closer to this kind of easier programming? Considerably short of solving the full Artificial Intelligence problem, I believe that we can do so by letting the language grow into specialized clusters, representing various categories of specialized user interest.

Accordingly, the language I envision has natural and easy to use extensions into what today we call applications. Rather than thinking of them as applications, however, we should think of these as specialized concepts, provided by additional software modules that are fully consistent with, and plug into, the basic MVC rails. Once a set of new modules is plugged in, it will manifest itself as a new set of user interface artifacts, new commands beyond the familiar ones of the basic language/system, and other new concepts that are familiar to the specialists using that module.

Thus, if one were interested in checkbook management, one would probably get from tomorrow's application vendors a module that would handle checkbook accounting and would therefore "understand" through its built in objects **totals**, **checkbook**, **category**, and **report** as new data interfaces; and **reconcile** as a new command. Likewise, if I were interested in accounting I would get the module that knows about **journals**, **posting**, **ledgers**, and **trial balances** as its primitive entities. Whatever I plug into the MVC rails, however, I am guaranteed that it will work gracefully and seamlessly with the basic MVC language/system and whatever else I already have plugged in. Using today's vernacular, but not today's distinct application worlds which are totally oblivious to each other's existence, this means that I should be able to easily call an information service with my communications module, and just as easily transfer the historical stock quotes that I receive through this action into my spreadsheet modules for analysis and then into my charting or report modules.

The issue here is not one of mere feasibility but rather of ease and convenience, and hence of productivity gain: We are not merely asking if there exists a spreadsheet program today that happens to do all of the above actions by design (there is one). Rather, we are asking that users be able to link easily **any independently developed modules** to do what the

users want to do, regardless of whether an application developer happened to think of doing the same thing. It is this property of the envisioned language to act in an integrative and cumulative way among numerous independent application modules, along with the ease with which new constructs would be developed on this base that would give the overall system its hoped for ease of use and power.

2.5 Conclusions

We need to get away from the current practice of simply covering up with the pretty colors of a user interface the debilitating complexity that has plagued programming since its inception. We should instead aspire to a more fundamental revolution in programming by inventing a new kind of radically different language.

Aimed at tomorrow's networked, multiprocessor architectures, such a language would integrate the entire computing environment of processor, communications, and input–output peripherals. It would simplify and incorporate as standard interfaces the commands and data representations that are common to most useful applications. It would easily extend its power via application modules to specialized domains, like accounting, design, planning, and music composition; and these extensions would be seen by users as natural additions to the standard interfaces that in many cases already represent the concepts of these higher-level activities. Finally, the modules developed for this language would be easily usable from other modules.

The programming process that would be used along with this language would be mostly in an out–in direction starting from the user interface. It would be accomplished largely through selection and easy modification of built-in or off-the-shelf objects. And it would employ substantial rapid and intelligent compilation and run-time decisions, leading to an easily tested and finished prototype at any stage of the development process.

Such a language used in such a way would blur traditional distinctions between programmers and users, among programming languages, operating systems and applications; and most important between user interface and program.

In effect the user interface would cease to exist as a separate entity and would become **the** language!