Guest Editor's Introduction

Multiparadigm Languages

Brent Hailpern, IBM

Have you ever gotten into a do-it-yourself construction project, and become stuck because you were missing a particular tool? You had three choices:

(1) go out and buy (or borrow) the right tool,

(2) use the wrong tool as best you could, or

(3) give up.

If you have not had this experience building a bookcase, baking a new recipe, or fixing your car, you most assuredly have while trying to write a large computer program. Admit it—you were programming in Pascal and wanted to use your computer's database query language, or you were using Lisp and you wanted to do complicated array manipulation, or you were creating an expert system in Prolog and needed a simple iterative loop or an abstract data type. When confronted by this problem in programming, you are limited to choices 2 or 3: program around the problem using the constructs provided by your language or give up.

A new class of programming languages and environments is being developed to help solve this problem. They do not restrict the programmer to only one *paradigm* (see box on p. 8); rather they are *multiparadigm* systems incorporating two or more of the conventional program paradigms. For example, the Loops system, described in the first article in this issue, combines features of the Lisp, functional, rule-oriented, and object-oriented paradigms. These multiparadigm systems are being created to give the programmer the right tool at the right time.

The beginnings

The most common multiparadigm system is the conventional operating system, which embodies several different programming languages. (The terms language and environment are defined in the box on p. 8. I use the term system to mean either a language or an environment.) Such large systems invariably require a linkage editor that combines object files from different compilers into one executable program. Linkage editors are difficult to use, often have their own obscure syntax, and may not work for all languages. The user is burdened with understanding the calling conventions (parameter passing and the like) of each of the languages involved.

As operating systems go, the Unix operating system provides for reasonable dynamic linkage editing through its pipes. Programs running under Unix are supposed to take their input from standard input and send their output to standard output. The pipe facility allows the standard output from one program to be tied to the standard input of another program, forming one large program out of a chain of small programs. If a program follows this convention, it can be written in any avail-

"The Three Gods of Construction," a four-foot by seven-foot acrylic-on-canvas painting by Sam Adelbai, is part of the permanent collection of the Institute of Culture and Communication's East-West Center in Honolulu. That city is also the site of HICSS-19, on which this issue is based.

The icons in Adelbai's work depict legendary figures of the Palau Islands in Micronesia that are similar—through migration, perhaps—to those of Hawaii and to the multiparadigm creatures of other cultures, six of which are represented in the multi-image at right and in the articles that follow.

Japanese: In the "Hell of Shrieking Sounds" (early Kamakura period, circa 1200) horse-headed guardians of a Buddhist hell chase priests who, because they mistreated animals, are condemned to eternal torment. Eugene Fuller Memorial Collection, Seattle Art Museum

Assyrian: Detail from alabaster relief, palace of Ashurnasirpal II, ninth century BC. Gift of Anna Bing Arnold, Los Angeles County Museum of Art

0740-7459/86/0100/0006$01.00 © 1986 IEEE

able language. Of course, this is a restricted form of multiparadigm system, because the composition is restricted to a linear chain of programs and the interface between programs is restricted to a sequence of ASCII characters.

Conventional operating systems are not adequate multiparadigm systems because of the strict separation of the different paradigms and the static nature of their linkage. Ideally, a multiparadigm system should allow language constructs from different paradigms to coexist within one program or module—just as while loops and procedure calls can coexist within a single Pascal program. Each paradigm of such a system should be able to refer to and depend upon services provided by the other paradigms.

Though multiparadigm systems provide tools to programmers, they are not the same as software development systems. Development systems emphasize tools to aid documentation, specification, and coordination between members of a programming team. Hence, they assist in the management of software development. Multiparadigm systems, on the other hand, aid the actual development of software.

Building a multiparadigm system

There are at least four ways to build a multiparadigm language. The first is to laminate together existing languages, for example, combining the syntax (and semantics) of Prolog, Lisp, and C into one system. An advantage of such a system is that users are already familiar with at least one component of the system, so they can start using the system quickly. A serious disadvantage is the complex semantics caused by unintended interactions between the component language constructs.

The second method is to augment an existing system with a new language construct, such as adding objects and methods to Pascal. This allows an existing body of software in the host language to be usable in the new system.

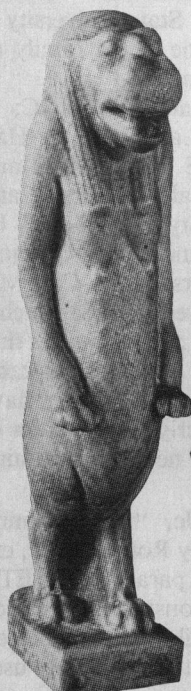
The third technique is to redefine an existing language in the context of new theoretical insights and goals. This method allows the designer to correct any undesirable features of the original language, add new paradigms, and provide a strong mathematical foundation for the result.

The fourth approach is to start from scratch: learn from the mistakes of others, develop a consistent formal basis, and build a new system. The advantages are consistency and elegance, but beware the effort needed to attract a user community.

This issue

The six articles of this issue of *IEEE Software* provide a glimpse into this growing field of software research. They are ordered generally from pragmatic to theoretical, but even the last (most theoretical) article has a running implementation. These six articles by no means represent all of the work being done: a special section of this issue contains summaries of nine other research projects in multiparadigm systems. The key word of this paragraph and of this issue is *research*. The techniques described here are still being investigated. Most likely some will fail, and only a few will succeed.

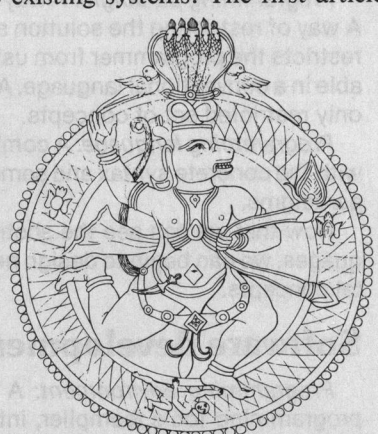
The first three articles of this issue discuss the addition of new paradigms to existing systems. The first article,



Egyption: Taueret protected women during childbirth and their offspring during infancy; Ptolemaic period, 664-30 BC. Gift of Mrs. William Leon Graves, Los Angeles County Museum of Art

Mayan: Coatlicue, mother god of the Mexicas.
Photo by Denice Killian

Greek-Roman: The powerful griffin—part winged eagle, part bold lion—is renowned for its courage.
Natural Science Picture Sourcebook, Janet Evans, ed., Van Nostrand Reinhold, 1984



Indian: Hanuman, the Hindu monkey divinity fathered by the wind, had the ability to fly and great speed.
Edward Moor, *Hindu Pantheon* (originally published in 1810), Philosophical Research Society, 1976

Some programming paradigms

Access oriented: The specification of side-effects or demons attached to the manipulation of variables [an extension of Loops].

Dataflow: Specifying the flow of data through a network of operations [VAL, ID, Lucid].

Data-structure-oriented: Approaching a solution by way of a single powerful data structure, such as lists (Lisp), arrays (APL), sets (SETL), and relational databases (SQL).

Functional: A pure mathematical specification of the solution to a problem, eliminating the conventional von Neumann model of memory and variables [pure Lisp, Backus's FP language].

Imperative: Sequential, block-structured commands with static scoping of variables [Fortran, Pascal, C].

Object-oriented: Grouping data into objects or abstract data types, where each object (or class of objects) has a set of operations (methods) to manipulate the data stored in that object [Smalltalk, Simula, CLU].

Parallel: Specifying multiple processes in the context of one processor or a distributed collection of processors [Modula-2, Ada, NIL, CSP].

Real-time: Specifying the constraints associated with physical objects such as robot arms, missiles, and analog devices [Arctic, AML, an extension to Lucid].

Rules-oriented: Specifying the constraints of the problem, rather than the algorithm for finding a solution [Prolog, OPS5].

The languages in brackets exhibit constructs that belong to the indicated paradigm. Some of the languages, especially the fourth-generation languages, are multiparadigm in that they incorporate features from different categories.

Language-induced paradigms

Programming language: The syntax and semantics (terminology and meaning) that defines a particular notation used to communicate a solution of a problem to a computer.

Programming paradigm: An abstract view of a class of programming languages that describes one means of solving programming problems. For example, both Algol and Pascal encourage the use of block-structured, sequential programs—hence they belong in the same paradigm.

Historically, languages came first and paradigms were abstractions based on features of existing languages.

Paradigm-induced languages

Programming paradigm: A way of approaching a programming problem. A way of restricting the solution set. By analogy, structured programming restricts the programmer from using all the unstructured constructs available in a conventional language. A paradigm allows the programmer to use only restricted set of concepts.

Programming language: A combination of one or more paradigms along with the concrete syntax and semantics to implement the notions of those paradigms.

Now that we can see the abstract paradigms behind the concrete languages, we can begin to design new languages based on these fundamental concepts.

Software development system

Programming environment: A programming language (or languages), programming tools (compiler, interpreter, linkage editor), and user interface (windows, menus, operating system shell, or command language).

Development environment: A programming environment plus tools for managing program development of complex systems by groups of people.

"Integrating Access-Oriented Programming into a Multiparadigm Environment" by Mark Stefik, Daniel Bobrow, and Kenneth Kahn, describes their experience with the Loops knowledge programming system at Xerox Palo Alto Research Center. Loops is a multiparadigm system to which they have added yet another paradigm: access-oriented programming.

The second article, "Extending the Scope of Relational Languages" by Henry Korth, describes how the relational database model can form the basis of a multiparadigm programming system. The work is an outgrowth of the ROSI project at the University of Texas at Austin, which has extended relational database concepts to an operating system interface.

The third article, "Toward a Real-Time Dataflow Language" by Antony Faustini and Edgar Lewis, discusses a syntax for expressing the requirements of real-time applications. This research at Arizona State University is an extension of the Lucid family of programming languages.

The next two articles, "FAC: A Functional APL Language" by Hai-Chen Tu of GTE Laboratories and Alan Perlis of Yale University and "Programming Styles in Nial" by Michael Jenkins and Janice Glasgow of Queen's University and Carl McCrosky of the University of Saskatchewan, describe two new languages that are descendants of APL. In each case, these two pieces of research have redefined the underlying semantics of APL to allow for new programming paradigms.

The final article, "Programming with Invariants" by Robert Paige, extends the set-based paradigm of SETL with an invariant construct. Instead of giving all the details of a complicated algorithm, this system allows the user to specify a simpler (inefficient) version of the algorithm and to give hints as invariant assertions. The system then combines the original program and the assertions to build a more efficient version of the algorithm. This scheme has been developed on the RAPTS system at Rutgers University.

The field of multiparadigm research is exciting and vital. The work is new enough that scientists and engineers from many allied computing fields are coming up with relevant ideas and systems. As this special issue of *IEEE Software* demonstrates, there already exists a collection of basic pragmatic research in the area. The time is ripe for some theoretical insights to tie together and generalize these existing empirical results. I believe that many more iterations of the experiment/theory cycle will come before this area is mature and well understood. □

Acknowledgments

This special issue grew out of a session at the 19th Hawaii International Conference on System Sciences, held in Honolulu during January 1986. I would like to thank all of the authors who submitted papers to that session and the referees that reviewed those papers. I must also thank Susan Knapp, Patrice Radwan, and Faith Compo for their invaluable assistance. This issue would not have been possible without the support and assistance of Bruce Shriver, editor-in-chief of *IEEE Software*. Finally, I would like to thank Lee Hoevel for his contributions to and support of multiparadigm language research at the IBM Thomas J. Watson Research Center.

Brent Hailpern is the senior manager of the experimental languages project at the IBM Thomas J. Watson Research Center, where he has worked since 1980. His research interests include concurrent programming, programming languages, and program verification.

He received the BS in mathematics from the University of Denver in 1976 and an MS and PhD in computer science from Stanford University in 1978 and 1980. Hailpern is a member of the ACM and a senior member of the IEEE.