There are several basic approaches to improving our ability to develop cost-effective high-quality software.

- Powerful language mechanisms improve programmer productivity. For instance, high-level languages such as Ada, as opposed to assembly languages, give programmers increased leverage in building large systems. Similarly, one-in one-out control constructs generally help reduce the errors that programmers make, thereby decreasing debugging costs.

- Programming paradigms — models of how programming should be done — help guide programmers in designing and structuring software systems. Paradigms define styles of thinking and programming that decrease the distance between the problem the programmer is solving and the program he or she produces. Different programming domains often demand different paradigms; for instance, a concurrent programming paradigm eases the development of operating systems. Functional and object-oriented programming are other examples of powerful paradigms.

- Careful definitions of software development processes help improve the resulting software product. The earliest attempt at such a definition — the waterfall model, where requirement definition, specification, implementation, testing, and maintenance were performed serially, with little feedback between phases — was not a realistic model of this process (although it was extremely useful in defining the notion of a software development process). More recent models — like rapid prototyping and Boehm’s spiral model — are showing increased utility.

- Reuse of existing software, avoiding original development to the greatest degree possible, is a well-recognized way to reduce software development costs. Mechanisms for finding and then sharing design and code are the current focus of numerous research efforts.

Motivation

Computer scientists have created numerous development tools for other disciplines, such as computer-aided design and computer-aided manufacturing. Only relatively recently, however, has the need for computer scientists to aid themselves been recognized. Using computers to support software development makes sense for three fundamental reasons.
First, a great deal of the effort in constructing software consists of menial tasks that can be automated — for instance, ensuring that object code has been compiled using the desired switches. Using the computer for such tasks allows the user to concentrate on the more challenging tasks, like system and algorithm design, that arise in software development.

Second, one of the most frustrating problems in software development is the difficulty of keeping code consistent with project information such as requirements and design documents, structure charts, and testing strategies. Using a computer permits much project information, which has historically been kept off line, to be kept on line, permitting environments to support the automatic and semi-automatic management of project information and code and the relationship between them. This information, once on line, is readily available to managers and other project personnel to help in coordinating project activities.

Third, the availability of powerful workstations with sophisticated input and output capabilities now makes it cost-effective to give each individual programmer the potential for interactive response times and rich user interfaces. These workstations make possible the construction of tools, such as dataflow diagram editors, that were simply not feasible in the mainframe world.

These three reasons arise, respectively, because of the ability of computers to compute tirelessly, to manage massive amounts of data, and to provide rich interaction with users.

**Topics**

Research and development topics related to environments can be viewed along three dimensions — activity, domain, and mechanism. The activity dimension defines the range of software development and maintenance activities that an environment addresses. The domain dimension defines the class of software that the environment is intended to support. The mechanism dimension defines the mechanisms that characterize the environment's internal and external organization.

**Activity dimension.** Software development and maintenance includes a wide range of activities — for example, scheduling, requirements definition, specification, design, implementation, testing, correction, retargeting, and enhancement. Environments can contribute to improved quality and productivity for any of these activities. Some environments focus on supporting a specific activity, while others focus on supporting a broad range of activities.

To date, the most successful environments have been those that focus on a narrow and related set of activities. For instance, DSEE's configuration manager facilitates the system composition and system generation activities. The more ambitious environments look to support the full range of software engineering activities. These broadly aimed environments appear to have the most promise for reducing the costs of software development, since they can help manage information that appears in many activities. Consider, for example, one scenario concerning activities that arises when a bug is reported from the field. Once the solution is found, the code must be changed, tested as a unit, and then run against a carefully defined set of regression tests. Then the change must be propagated, perhaps through the design specifications and documentation and certainly through other versions of the system. An environment focused on debugging, for instance, cannot hope to help with most of these chores. This limitation of narrowly focused environments...
leads us to pursue more broadly targeted environments. Despite the promise of these broader environments, however, no single instance has yet demonstrated, at least in any commercially clear-cut way, the ability to manage information successfully across most activities.

**Domain dimension.** Just as it is no longer generally accepted that there is a single programming language that is suitable for all tasks, it is not possible to construct a single environment that is suitable for all domains. Appropriate design techniques and debugging support, among others, must often vary from domain to domain; for example, a debugger for a functional language may not need to show "current" values of variables, since their state cannot change. Additional knowledge about a domain can always be used to improve the support an environment gives to its users.

The original (and still a common) approach to constructing environments for different domains is to handcraft each one. The benefit of this approach is that embedding domain-specific knowledge is straightforward. Good examples of successful handcrafted environments include Cedar, Xerox PARC's single-language environment for building distributed systems, and the Cornell Program Synthesizer, a syntax-directed editing environment for introductory programming courses. The problem with hand-crafting environments is the development expense. Constructing environments is costly, closer to the costs of operating system development than to compiler development.

But even diverse environments share many characteristics, often permitting reuse of abstractions and implementations. Environments for Prolog programs and environments for parallel programs, for instance, might well share windowing systems and text editing capabilities. Similarly, environments for large-size C programs and environments for Jackson System Design might share compilers, linkers, and documentation templates.

Environment generation projects ease the construction of a family of environments. The key to this approach is the explicit separation of an environment-independent kernel, which is shared across all environments in the family, from a set of environment-dependent definitions, which specialize the kernel for particular domains. In Gandalf, one of the first environment generation efforts, the kernel defines a model of structure-oriented editing (where users edit in terms of structural constructs, such as if-then-else statements and headers of mail messages), while the environment-specific information defines the actual structures of the environment, along with the semantics of these structures. Environment generation efforts like Arcadia broaden the environment-independent kernel to include an interpretation engine for environment-specific "process programs," which explicitly define the software process model desired for a particular domain.

The trade-off between the cost of developing environments and the support the environments provide to the user is not yet settled. The need for individual groups, or even individual users, to customize environments to their specific needs is evident. In some areas, like language syntax and semantics, suitable sharing is possible. The degree to which sharing and reuse is feasible for other aspects of environments is cloudier.

**Mechanism dimension.** Carefully designed and chosen mechanisms facilitate the development of an environment for a given domain and set of activities. To implement an environment, given a basic understanding of the requirements, the implementor must consider appropriate external and internal structures.

External structures designate the style of interaction between the environment and its users; for instance, if the domain of the environment is novice programmers, a reliable and extremely flexible help facility might be needed. Internal structures reflect on the way in which the environment is composed; for instance, Unix tools are generally built around the notion of standard input/output, which helps compose tools in a batch-like fashion, but not in the interactive style needed by many environments.

In many cases, only a fine line is drawn between mechanisms that support internal and external structuring. Consider the Pecan system, for instance. A key notion of Pecan is that users should be allowed to see their program in several fundamentally different ways. For instance, a user might see a program either as lines of text or else as a Nasi-Sheneiderman chart. This notion, however, places significant demands on the internal mechanisms, which must support techniques for defining alternate views of viewing a program as well as techniques for keeping the views consistent (both visually and internally).

There are three key things to note about environment mechanisms. First, there is an extremely close relationship between internal and external mechanisms; in general, internal mechanisms are developed in response to external mechanisms and needs. Second, as has been demonstrated in operating systems, it is beneficial to separate mechanism from policy; that is, a mechanism should give the maximum freedom possible to the implementor that uses the mechanism. Third, to reduce the long-term costs of developing environments, it is necessary to develop mechanisms that are as ubiquitous as possible, for use in environments of the same family as well as, perhaps, in others.

**Current status**

Research and development on environments have achieved some successes and must be noted for some failures.

**Activities.** Environments that focus on a relatively narrow range of software development activities have been successfully constructed and used. Most notable are environments (like Smalltalk-80 and Interlis) that concentrate primarily on the programming-in-the-small activities of editing, translating, executing, and debugging. Environments that support early activities in the software life cycle — such as IDE's Software through Pictures — are appearing now as well. Reasonably successful environments have been produced to support virtually every software development activity.

However, it has not yet been demonstrated that the full range of activities can in fact be supported effectively within a single integrated environment. Efforts in this dimension — such as Gandalf, which supports programming-in-the-many (the cooperation, communication, and coordination of multiple programmers), programming-in-the-large (the composition of modules into systems), and programming-in-the-small — have produced prototypes rather than production systems. Although these prototypes show promise, full working environments are needed to substantiate the claims made for broad-based environments.

**Domains.** Environments for some domains have also been successfully built and used. Cedar's intention to support exploratory distributed-systems programming has largely been met, for example.
We cannot yet develop environments for widely varying domains in a cost-effective way. However, we are making strides in this direction due to our initial experiences with separating environment-dependent aspects from environment-independent aspects. We are broadening the domains we can accommodate by carefully, and usually formally, characterizing various aspects of software development. Examples of such characterizations include abstract syntax trees, which drive structure-oriented editing environments, and program dependence graphs, which support debuggers and optimizers. As we increase our ability to deal with an expanded range of domains, we will be driven to consider environments that simultaneously support multiple domains. Work on multiple domains at the language level should provide a basis for multi-domain environments.

Mechanisms. A variety of mechanisms and classes of mechanisms have formed the basis of many successful prototype and production environments. Examples include both incremental algorithms, such as Reps's attribute grammar evaluation scheme, that provide one of the key underpinnings of interactive environments and also advances in user interface technology, such as those that arise from the early experiences with Smalltalk, that provide a solid basis for much of the external appearance of environments.

There are several specific areas where additional mechanisms more appropriate to environments are needed. One of these areas is database support, as Bernstein has so clearly shown. Rich type models, multiple representations, and versioning are just some of the characteristics that existing database and file systems do not support in a suitable way for environments. Another area is tool integration. Constructing and evolving tools in interactive environments is extraordinarily difficult, since the interactions among the tools must generally be dealt with explicitly. Although progress is being made by several research groups, no model as simple and ubiquitous as Unix pipes has yet been developed. (Some representations of software entities, such as abstract syntax trees and program dependency graphs are widely used in environments; however, general tool integration mechanisms that allow these representations to be easily glued together into environments are still lacking.) Taken together, these mechanisms form the basis for an environment.

Software Engineering Opportunities

RADE*, a Micrognosis product, is an integrated workstation that is designed to put the active traders' need for fast breaking information at their fingertips. Leading banks, brokerage firms, insurance companies use Micrognosis systems as their link to the world of financial information.

We have exciting opportunities for qualified professionals in the following areas:

Software Manager
Product Engineering
You will be responsible for the supervising of project teams during all phases of software enhancements.
An ability to set and meet full project lifecycle goals and objectives in a highly technical environment is required. We expect you to have a solid knowledge of "C" and PLM operating under UNIX* in an on-line environment.

Software Manager
Research & Development
You will direct project teams involved in the development of state-of-the-art, microprocessor-driven software for the Financial Services Industry.
You must possess strong analytical, conceptual and technical management skills and a knowledge of "C", PLM and data communications under UNIX and VMS.

Group Leader
Software Systems Test
You will be responsible for the verification, validation and evaluation of product and system releases.
You must have at least 6-8 years experience with a previous supervisory background, and experience in microprocessor-based systems with knowledge of VAX and microVAX, UNIX-based with "C" experience a plus.

Senior Systems Test Engineers
Preferably, you should have 5-7 years experience in the design, implementation and test of microprocessor-based or VAX-based software systems.
A knowledge of various software systems test methodologies is required along with a knowledge of "C" and data communications under UNIX would be a plus.

Group Leader
Software Test Tools & Simulation
To design, develop and customize test tools and simulation for microprocessor-based and VAX-based real-time distribution systems.
For this senior level position, with group leader responsibilities, you must possess 7-10 years experience and a knowledge of VAX, UNIX and "C."
For all of the above positions a BSCS or BSEE or its equivalent is required or preferred.

IT'S ABOUT TIME!

Micrognosis can offer you an outstanding salary, comprehensive benefits and exceptional opportunities for professional growth.
Join a company that can provide real career momentum. It's time to send your resume, indicating position of interest, salary history and requirements in confidence to: Vito Santoro, Manager of Staffing & Development, MICROGNOSIS, Inc., Dept. IEEC 101, 100 Saw Mill Road, Danbury, CT 06810. An equal opportunity employer.

Applying technology to trading

*UNIX is a trademark of AT&T Bell Laboratories

MICROGNOSIS, INC.

November 1987
architecture, which provides a platform in which constructing environments is qualitatively simpler than is currently possible. Research on individual mechanisms and on complete environment architectures is necessary.

Evaluation. Perhaps the biggest failing of environments research and development to date is the general lack of scientific evaluation of existing environments. Evaluation approaches and actual evaluations are beginning to appear, but relatively little effort has been given to this undeniably fundamental subject.

Despite listing this as a shortcoming, it is important to note that, for several reasons, evaluation of environments is difficult. First, useful qualitative analysis of programmer productivity and software quality improvements are hard to produce; similarly, most quantitative analyses do not, in and of themselves, provide sufficient information for proper evaluation. Second, to effectively evaluate an environment, one must have a suitably large user community; recruiting such a community is difficult or impossible in many situations, especially where the environment is intended for multi-person multi-year software system development, where the evaluation costs would be excessive. Third, the design space for environments is enormous; hence, comparative evaluation is unwieldy since so many experimental variables change from environment to environment.

An understanding of the importance of evaluation of environments is essential before the field can actually produce powerful and effective environments.

The articles in these companion issues address but a portion of these problems. Many of the articles solve important, but still limited, cases of the problems. These incremental and consistent strides towards solving the key problems in environments show that the field has significant potential for meeting its goal of improving the productivity and quality of software developers and of software.

Acknowledgments

Bruce Shriver, editor-in-chief of Computer, has helped significantly with these companion issues. We appreciate the efforts of the authors of the articles that were, and were not, accepted. We also thank the many referees who put in valuable time to help us select the best possible articles. Special thanks go to the administrative staff — in particular, Margie Ramsdell of the University of Washington and Kathy Germana of SUNY at Stony Brook — who helped manage the many details that had to be handled. Everyone involved — including editors, authors, referees, and staff — did a great job while working under time pressures that were always great and often unreasonable.

References


Peter B. Henderson, who had primary responsibility for the articles appearing in IEEE Software, is an associate professor in the Department of Computer Science at the State University of New York at Stony Brook. His main areas of interest are software engineering, programming environments, and computer science education.

Henderson received his BS and MS degrees in electrical engineering from Clarkson College and his PhD from Princeton University. His address is Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.

David Notkin had primary responsibility for the articles appearing in Computer. He has been an assistant professor in the Department of Computer Science at the University of Washington since September 1984. He was a charter member of the Gandalf project at Carnegie Mellon University and received an IBM Faculty Development Award in 1985. Currently, his research interests include software development environments, extendable software systems, programming-in-the-large, structure-oriented editing environments, interactive programs, parallel programming environments, and heterogeneous computer systems.

Notkin received his ScB degree from Brown University and PhD degree from Carnegie Mellon, both in computer science, in 1977 and 1984. His address is Dept. of Computer Science, FR-35, University of Washington, Seattle, WA 98195.