ACCOMMODATING UNCERTAINTY IN SOFTWARE DESIGN

Recognition that most software is domain dependent (DD) is extremely important because the most commonly used software life-cycle models are not adequate for DD software. The nature of DD software, and the need to manage its life cycle effectively, calls for a new approach to software design and the implementation of software development environments.

RICHARD V. GIDDINGS

A review of current literature would lead one to believe that the "software crisis" is a recent development. Such is not the case. There has always been a software crisis.

Software development techniques, which have matured little, require large amounts of highly skilled labor. Because the necessary labor has rarely been available, personnel with marginal skill levels have been, and are increasingly, in high demand.

The impact of the ongoing shortage of skilled labor is staggering. For example, it is estimated that up to 90 percent of the data processing intellectual effort in a large corporation is devoted to maintenance (namely, redesign, reprogramming, and error correction [6]).

Successful resolution of the software crisis requires a significant change in the manpower-intensive nature of the development process. It must be based on a redefinition of the process rather than on further value engineering. Such a redefinition must start by examining the basic assumptions about the nature of the software development process.

It has been recognized that software is not homogeneous, but only recently have software classifications begun to appear that are based on the relationship of the software to the environment within which it operates [5]. These classifications, one of which is proposed in this paper, provide an improved model for explaining program dynamics and developing life-cycle management strategies.

A portion of this paper was originally published in the *Scientific Honeywellet* 4, 2 (June 1983), 11–13.

SOFTWARE MODELS AND LIFE CYCLES

Perhaps the most commonly used model of the software life cycle was developed by Boehm [2] and is shown in Figure 1. At a high level, the development process is viewed as a progression from problem definition to implementation to maintenance.

For many interesting classes of software, Boehm's life cycle does not adequately model the development process. Consider the following scheme that classifies software according to the way in which the universe of discourse (the class of problems to be computed) and the software interact.

Domain Independent (DI)

This class of software is distinguished by the independence of the problem statement and the universe of discourse (that is, solutions need to be verified but not validated). Figure 2 provides a model for this type of software.

For this class of software, the development process can be described as a search for one of many "good" solutions. The essential problem is proving that one has in fact obtained a solution (verification).

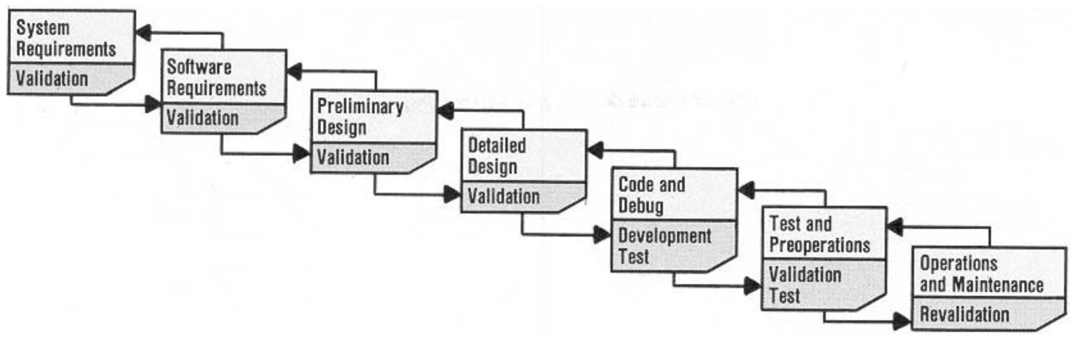


FIGURE 1. Boehm's Software Life Cycle

Examples of this type of software include numerical algorithms or, from a practical point of view, software developed under a contract with predetermined specifications and no ongoing responsibility for the developer other than bug fixing.

Domain Dependent Software (DD)

Two distinct types of software make up the class of DD software: experimental (DDEX) and embedded (DDEM). DDEX software is characterized by an intrinsic uncertainty about the universe of discourse (see Figure 3). The development process is embedded within a search for knowledge about the universe of discourse. The essential problem is producing software useful for testing a hypothesis or exploring unknown characteristics of the universe.

Examples of this class of software are models being used as vehicles for conducting research to discover information about a universe of discourse. In such efforts, one is trying to identify necessary data, data collection constraints (for example, accuracy or frequency), relationships, and systems dynamics.

The use of DDEX software may eventually lead to the development of a specification for software with other uses (for example, an economic model that can be used to improve decision making). However, that software would not be DDEX.

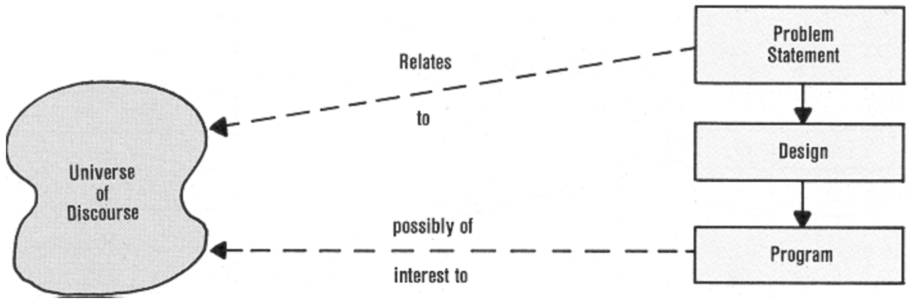
A model for DDEM software is shown in Figure 4. This software is characterized by interdependence between the universe of discourse and the software. The use of the software may change both the form and the substance of the universe of discourse and, as a result, the nature of the problem being solved.

Examples of DDEM software include business systems, office automation systems, software engineering systems, design automation systems, and successive generations of a large-scale operating system. In each of these, the development process is a search for a "good" problem statement. The essential difficulty lies in anticipating the impact of likely changes in the universe of discourse resulting from the introduction of the software.

An interesting phenomenon often associated with this type of development is that the introduction of the software serves as a catalyst for changes in its environment that far exceed those anticipated by the software designers. For example, some experts believe that 80 percent of the gain from office automation will result from concomitant factors such as work reorganization or job redesign, whereas only 20 percent will be derived directly from the application of advanced automation technology [4].

Domain Dependent Software Life Cycle

Historically, software methodologies have focused on programming techniques. Today, many focus on design. The few that address the entirety of Boehm's software life cycle rest on the a priori assumption that the designer has, or can obtain, a detailed understanding of the problem and can implement a solution and move on to another project leaving a rather mundane aspect—maintenance—to others.



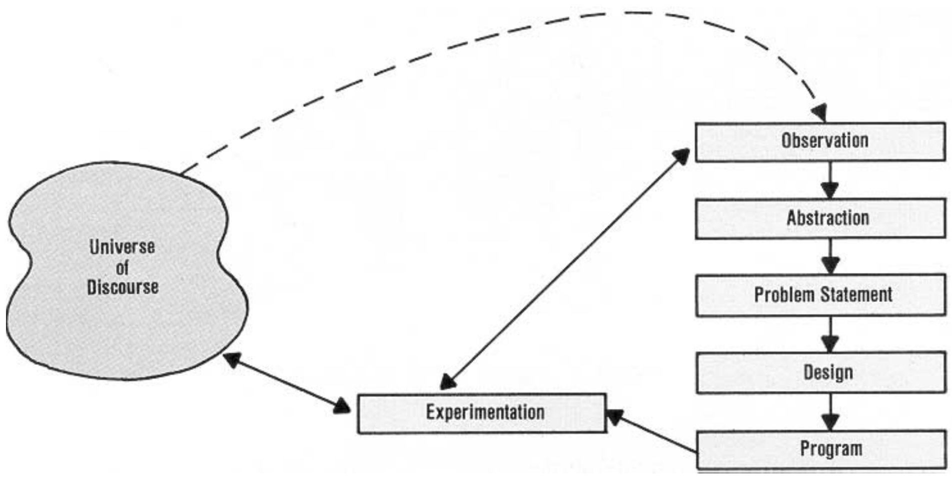


FIGURE 3. Domain Dependent Software—Experimental

For DD software, the above is only trivially true (that is, the designer knows the current problem statement but does not know the relationship between that problem statement and a problem statement that leads to a useful solution). Rather than implementing a solution, one is really refining a sequence of imperfect prototypes over an extended time (see Figure 5). For this reason, conducting experiments to validate the problem statement and to provide feedback for successive prototypes is an essential part of the development process.

Before proceeding, it is worth noting that one very common event is not explicitly represented in Figure 5. At some point, a prototype typically becomes useful to others besides the developers or experimenters. When this occurs, the prototype can be made available as a product or the specifications for the prototype can be used as the problem statement for a DI software development effort to produce a reengineered product. If the prototype is made available as a product, a "frozen" copy of the software enters a maintenance phase limited to bug fixing.

Treating a product as a "spin-off" from the software development cycle with maintenance limited to bug fixing is useful for three reasons. First, it allows one to distinguish between bug fixing and program evolution. These two distinct activities have been traditionally clumped under the term, *maintenance*. Second, having made such a distinction, one can contrast management procedures designed to insure the short-term stability of a product with those designed to cope with a long-term process where continuing change is intrinsic. Third, one can conduct field evaluations as a check on the reliability of experimental validation efforts.

DOMAIN DEPENDENT SOFTWARE DEVELOPMENT LIFE-CYCLE IMPLICATIONS

Much of the current software crisis is a result of not recognizing and not managing the empirical, ongoing

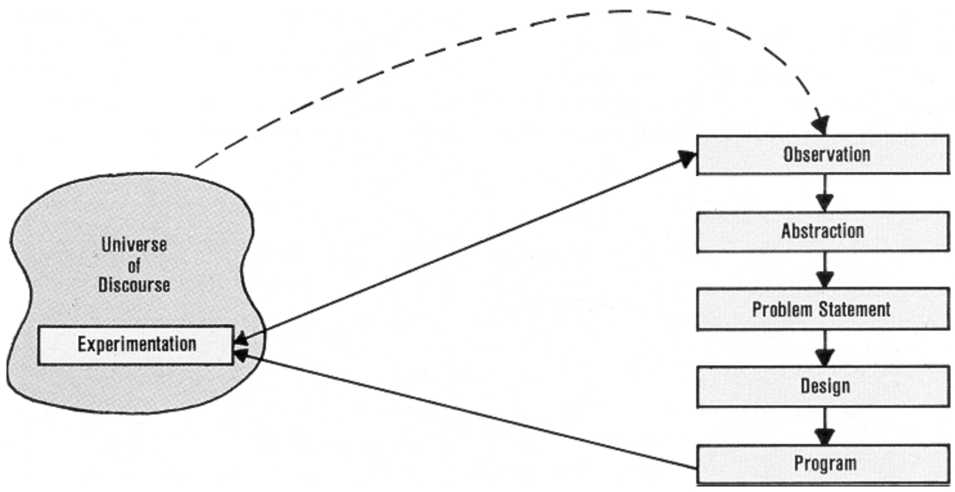


FIGURE 4. Domain Dependent Software—Embedded

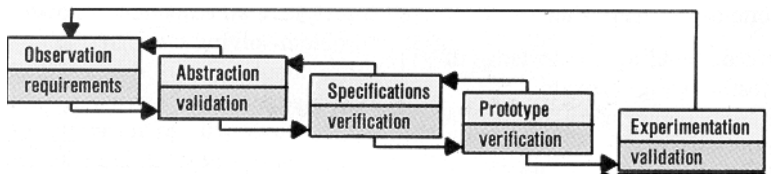


FIGURE 5. Domain Dependent Software Life Cycle

nature of DD software development (that is, using inappropriate DI development procedures). In fact, when one considers the perceptual and communication problems inherent in the software development process, DI software development may be quite rare.

Because the development of DD software is a process of refining prototypes, the basic management tradeoffs are both the total life-cycle cost and the time necessary to produce successive prototypes. Clearly, to be optimal, any methodology based on designing each of a sequence of prototypes from scratch will require a minimum amount of effort to build each prototype. Also, hierarchical system designs do not lend themselves to program evolution [5].

Using the tools and techniques available today, prototype development can be accomplished with very little effort for some types of problems. The trend toward using very high-level programming languages will increase the number and type of problems that can be effectively managed by building successive prototypes from scratch. However, for the foreseeable future, we will be faced with the recurring requirement of reducing the total life-cycle cost by increasing the probability that work invested in one prototype can be easily carried forward to succeeding prototypes.

One approach to protecting the investment made in any given prototype is to collect modules into libraries for reuse—either modules resulting from the structured design for each prototype or modules developed for other projects that happen to be accessible. However, module libraries of this type (that is, collections of modules that happen to be available) have been around since the 1950s and have failed to offer significant advantage.

The essential problem is that the individuality and creativeness of a software designer are reflected in the hierarchical decomposition of a problem statement. There is no reason to believe that modules obtained through such a process would ever be directly applicable to another, or a succeeding, development effort. The time and effort spent in searching for and modifying modules that are "close" to the desired functionality typically outweigh the cost advantages of reusing code.

Other problems deal with poor organization of the module libraries. For example, a useful library must provide quick and easy access to modules that might provide the necessary functionality, must specify clearly module function and implementation constraints, and must ensure that modules are "high quality."

As an alternative to the module library, Wasserman and Belady [8] proposed the establishment of a "software inventory." Later, Belady [7] proposed the concept of "evolved software." Parnas [7] described "designing software for ease of extension and contraction." I have proposed the idea of "component software" and, based on experience gained with the REAP system [3], the idea of a "component software development environment."

In each of the above proposals, the essential idea is to design software components for reuse. The internals of the components (parts or building blocks) are to be "unknowable" to the user, thus allowing the component designer the freedom to experiment with implementation strategies.

Each of these approaches offers the potential for overcoming the difficulties associated with the typical module library. However, significant problems remain.

The first difficulty with designing software components for reuse is identifying the components that should be provided. There is intuitive appeal to the idea that, for a given universe of discourse, there should be "optimal" sets of components. These sets are optimal in the sense that the cost of the component library (that is, component development cost; library development and operational costs; and cost of searching for a component, verifying its suitability, and incorporating it into a design) plus the total cost for a sequence of prototypes is minimized. The second difficulty is determining an effective environment within which to develop prototypes using components.

The next section proposes a method for addressing these two difficulties.

DOMAIN DEPENDENT SOFTWARE DEVELOPMENT ENVIRONMENTS

Before proceeding, it is necessary to establish definitions for a number of commonly used terms.

- A *task* is a narrowly focused activity usually performed by a single worker.
- A *tool* is something that facilitates the performance of a task.
- *Mechanization* is the use of tools.
- A *problem-solving environment* is an integrated set of tools used to accomplish a function.
- *Automation* is the use of that class of systems that requires no human intervention other than at initiation and at termination.
- A *problem-solving strategy* is a procedure followed by a human in obtaining some "end."

With these definitions, one can observe that

- Mechanization requires embedding a knowledge of tasks into the tools. Most existing software development environments focus on mechanization of software development tasks.
- Problem-solving environments require embedding a knowledge of ends, tasks, and problem-solving strategies into integrated systems (that is, they are "knowledge-based systems" or "expert systems").
- Automated systems require embedded knowledge of an end and an algorithm for achieving the end.

For all types of software, we are already seeing an increasing focus on the development of problem-solving environments because mechanization does not produce the productivity advantages of an integrated set of "intelligent" tools and, except for trivial cases, automation is not feasible.

Problem-Solving Environments and DD Software

The economics associated with managing the DD software life cycle are significantly different than those associated with the DI software life cycle. When one considers life cycles that typically extend over many years and may result in expenditures of millions of dollars, long-term cost tradeoffs are available that are often in conflict with the short-term nature of the DI development process.

The most important of these cost tradeoffs is the front-end development of problem-solving environments and management procedures designed to minimize the DD life-cycle cost. Cost reductions can be achieved in three complementary ways. First, a problem-solving environment can be designed to minimize the work invested in a sequence of prototypes (that is, an environment can be designed to increase the probability that effort invested in one prototype will be effectively utilized in successive prototypes; hence the cost can be prorated over a larger base). Second, a problem-solving environment can reduce the cycle time—the time from requirements analysis to experimental validation of a prototype. This reduced cycle time results in more effective products and reduced personnel costs. Third, a problem-solving environment can be designed for use by personnel with marginal data processing skills, thus conserving highly skilled manpower [3].

At present there are three forms an effective problem-solving environment may take:

(1) generic environments applicable to all software development,
(2) special-purpose environments for use with a specific universe of discourse (that is, a specific class of software development problems), or
(3) extensible environments that not only provide support for the development life cycle but are simply "extended" to produce prototypes and products.

It is my opinion that the development of generic environments, the bulk of current efforts, will not yield significant results. A problem-solving environment must have an embedded knowledge of ends, tasks, and problem-solving strategies; at a generic level, knowledge about software development can be embedded, but the amount of knowledge about any other universe of discourse will, by necessity, be small.

Special-purpose problem-solving environments have been shown to offer cost advantages and to hold significant potential for DD software. For example, the REAP system was built using a special-purpose environment designed specifically for use in building environmental systems. Overall development cost was reduced from an estimated \$8 million to an actual \$1.6 million [3]. Productivity of software development personnel was the equivalent of 1300 lines of production FORTRAN code per person-month at a cost of \$2.84 per line of operational code. The end product, consisting of over 360,000 lines of FORTRAN code, is maintained by one person working part time at this task.

Extensible environments are interesting to think about; however, there are more questions than answers about cost and effectiveness. They are mentioned here primarily because the approach to problem-solving environment design described in the next section may also provide an approach to the design of extensible environments.

The Design of Problem-Solving Environments for DD Software

At this point, we need to return to the two issues left hanging earlier: how do we identify the set of components to be provided, and what is an effective problem-solving environment for using components? We can restate these issues a little more precisely as follows:

- Given a universe of discourse, how does one identify an "appropriate" set of components?
- Given that one has an "appropriate" set of components, what is the effect of changing the universe of discourse? (Or, given a set of components, what is the effect of adding or deleting components?)
- If there is a procedure for determining both of the above, does it make a difference how one selects a universe of discourse? (For example, if the universe of discourse I am interested in is the "class of all business problems," but I anticipate dealing also with the "class of all integrated circuit design problems," what is the effect of selecting a universe of discourse that is the union of those two sets?)
- How does answering the above affect the design of problem-solving environments?

Using concepts from formal mathematical model theory, one can formalize the first two of the above quite easily. In particular, the first two questions deal with "closure" and "consistency" of mathematical models.

Although it may not be obvious, the third question can be viewed primarily as a human factors tradeoff. Formal models can be easily developed for universes of discourse ranging from the class of all computable problems to a single, simple problem statement. As we shall see later, selection of the universe of discourse affects our ability to produce a model that is "easy to use" (in a sense defined later).

The fourth question boils down to, "If one goes to the effort of developing a formal model for a universe of discourse, can a problem-solving environment be developed that implements the formal model in a straightforward way and would such an implementation be usable?" The answer to this question is *yes.*

A methodology based on a formal modeling approach consists of four phases:

(1) describing the universe of discourse—an in-depth analysis of tasks, work flow, end-user behaviors (namely, problem-solving strategies), and organizational goals and objectives;
(2) developing a formal model;
(3) implementing the model;
(4) developing applications.

This approach is simplified by the following hypotheses:

(1) Human problem-solving strategies and behaviors can be represented as algorithms, which in turn can be represented as recursive functions (Church's thesis).
(2) The set of an individual's problem-solving strategies and cognitive processes, although dynamic, has a small cardinality.
(3) For any given class of problems, the cardinality of the set of human behaviors used to solve problems is small.

The first, a thesis that has held for 40 years, allows one to develop a formal model of the desired problem-solving environment and assures that it can be implemented in a straightforward manner. The second two, basic premises of cognitive psychology that have been studied since the turn of the century, allow one to build "easy to use" systems—easy to use in the sense that they

(1) minimize the number of end-user steps required to achieve a solution, and
(2) minimize impedance (that is, the steps a user is required to follow reflect both what tasks the user believes to be important and the order in which they should be performed).

It follows immediately that ease of use is not an absolute measure. Rather, it is a measure with respect to a single problem domain and a specific class of end users. It also follows that the complexities associated with designing an "easy to use" system are related to human factors and cognitive psychology more than to data processing.

Several advantages are provided by formally modeling the problem domain and the behaviors of users that interact with that domain. First, a formal model provides the information necessary to build easy-to-use systems. Second, a formal model provides a natural way to identify and design "code" for reuse. Third, verification is simplified (that is, because the approach is constructive, one needs only to verify the correctness of the composition and, independently on a one-time basis, the model itself). Fourth, a formal model offers the potential of developing hardware and software architectures optimized for the universe of discourse.

For example, the nature of a "program" lends itself to the development of very high-level programming languages. Also, in my opinion, Wilner's [9] novel hardware architecture (particularly well suited to VLSI technology and avoiding the "Von Neumann bottleneck") would provide a good vehicle for the implementation of formal models.

Nested Development

The development of a problem-solving environment for DD software development adds confusion to the issue of life-cycle management. Because one is normally conducting two development efforts concurrently (that is, both the problem-solving environment and the application are DD software), there is a nesting of life cycles (see Figure 6). This nesting suggests the idea of extensible problem-solving environments and an approach to the design of that type of an environment (again, see Figure 6).

CONCLUSION

DD software development is an empirical, ongoing process. As such, it is not surprising that the front-end implementation of a problem-solving environment offers a high return-on-investment opportunity as well as a means for conserving skilled data-processing personnel and increasing product effectiveness.

Reusing tested, verified code is essential if software productivity is to be significantly improved by increasing our ability to effectively carry the investment in

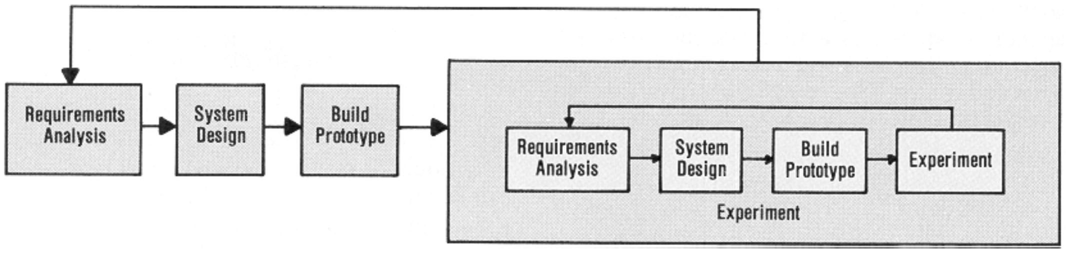


FIGURE 6. A Nested Development Cycle

one prototype forward to succeeding prototypes. However, reusable code will not be an accidental spin-off of current practices. Code with a high probability for reuse must be identified and designed for reuse, and an environment that encourages reuse must be created.

Special-purpose and extensible problem-solving environments (with their productivity improvements, better ability to conserve skilled manpower, and higher potential for reusing code) will be increasingly emphasized over generic problem-solving environments. The question of whether an optimal set of extensible problem-solving environments could be defined and extended to create special-purpose problem-solving environments will require a significant amount of research to resolve. However, if such a set were identified, software development costs could be significantly reduced.

Issues about the design of problem-solving environments can be formalized by means of model theory. Models derived through such a process could be directly implemented and hold the potential of offering alternatives to current practice.

REFERENCES
1. Belady, L.A. Evolved software for the 80s. *Computer* (Feb. 1979), 79–82.
2. Boehm, B.W. Software engineering. *IEEE Trans. Comput. C-25* (Dec. 1976), 1226–1241.
3. Giddings, R.V. A graphics-oriented computer system to support environmental decision-making. In *Computer Graphics and Environmental Planning*, E. Teicholz and B. Berry, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1983.
4. Hammer, M. What is office automation? Off. Autom. Memo 12, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Jan. 1980.
5. Lehman, M.M. Programs, life cycles, and laws of software evolution. *Proc. IEEE 68*, 9 (Sept. 1980), 1060–1076.
6. Martin, J. What to plan for to manage the future of your data center. *Can. Datasyst. 9*, 3 (Mar. 1977), 28–32.
7. Parnas, D.L. Designing software for ease of extension and contraction. *IEEE Trans. Comput. SE-5*, 2 (Mar. 1979), 128–137.
8. Wasserman, A.I., and Belady, L.A. Software engineering: The turning point. *Computer* (Sept. 1978), 30–39.
9. Wilner, W.T. Recursive Machines. Rep. P.800054, Xerox Palo Alto Research Center, Palo Alto, Calif., June 1980.

CR Categories and Subject Descriptors: C.O [**General**]: *system architectures*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries, user interfaces*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.9 [**Software Engineering**]: Management—*life cycle, productivity*; D.2 [**Software Engineering**]: Miscellaneous—*rapid prototyping, reusable software*

General Terms: Design, Economics, Human Factors, Languages, Management
Additional Key Words and Phrases: problem-solving environments

Author's Present Address: Richard V. Giddings, Manager, Local/Office Systems, Corporate Information Management, Honeywell Inc., Honeywell Plaza, Minneapolis, MN 55408.