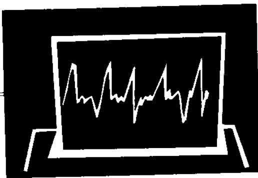


STATUS REPORT: SOFTWARE REUSABILITY

The problem is not a lack of software reuse, but a lack of systematic reuse. Industry will achieve big payoffs only if this can be changed. This article assesses where reuse technology stands.

RUBÉN PRIETO-DÍAZ
Reuse, Inc.

The notion of reusability has been around since humans began to solve problems. When we solve a problem, we try to apply the solution to similar new problems. If we find that only some elements of the solution apply, we adapt it to fit the new problem. Proven solutions, used over and over to solve the same type of problem, become accepted, generalized, and standardized.

Formal generic mathematical models are good examples of successful reuse because they can be applied to solve specific problems across several engineering fields or domains. The successful abstraction of a class of problems in the mathematical domain, its solution in that domain, and the ability to apply that solution to specific instances in other domains not only captures the essence of the reuse problem but shows the tremendous potential of reuse at high levels of abstraction.

However, reuse can also be effective at the artifact level, where elements can be standardized and generalized. Movable type is a classic example of successful artifact reuse.

In software engineering, generic algorithms are the equivalent of reusable high-level abstractions; subroutines, macros, functions, and objects are examples of reusable artifacts, or components.

The problem we face in software engineering is not a lack of reuse, but a lack of widespread, systematic reuse. Programmers have been reusing code, subroutines, and algorithms since programming was invented. They also know how to adapt and reverse-engineer systems. But they do all this informally.

The reuse research community is focusing on formalizing reuse because it recognizes that substantial quality and productivity payoffs will be achieved only if reuse is conducted systematically and formally. The history of reuse is characterized by this struggle to formalize in a setting where pragmatic problems are the norm and fast informal solutions usually take precedence.

REUSE HISTORY

Software reuse is the use of existing software components to construct new systems. Reuse applies not only to source-code fragments, but to all the intermediate work products generated during software development, including requirements documents, system specifications, design structures, and any information the developer needs to create software.[1]

Dough McIlroy introduced the concept of formal software reuse in what has become the field's seminal paper.[2] McIlroy proposed an industry of off-the-shelf, standard source-code components and envisioned the construction of complex systems from small building blocks available through catalogs.

McIlroy's idea was key to the software factory concept. Systems Development Corp., which registered the term "software factory" as a trademark in 1974, established a well-defined set of procedures for a consistent and repeatable software process, but reuse was implicit at best.[3] In SDC's software factory, programmers were expected to reuse whatever they could, from code segments to previous experience, but reuse was not defined in the process. The SDC experience was used as a basis for future software factories, especially in Japan, where formal reuse processes were later defined and integrated.

Also in the mid-1970s,

0740-7459/93/0500/0061/$03.00 © IEEE

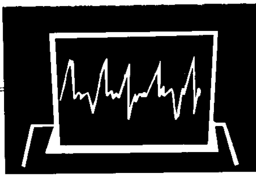


TABLE 1
FACETS OF REUSE

By substance	By scope	By mode	By technique	By intention	By product
Ideas, concepts	Vertical	Planned, systematic	Compositional	Black-box, as-is	Source code
Artifacts, components	Horizontal	Ad-hoc, opportunistic	Generative	White-box, modified	Design
Procedures, skills					Specifications
					Objects
					Text
					Architectures

Robert Lanergan began a reuse project at the Raytheon Missile Division that is now considered the first case of formal reuse at the organization level.[4]

Interest in reuse spread through academia in the late 1970s. The best collection of reports from the several research projects initiated during this period is the proceedings of the First International Workshop on Reusability in Programming.[5] This workshop is also notable for the consensus reached by attendees that reuse should be pursued at several levels of abstraction and that it involves several nontechnical issues of significant relevance.

In the 1980s, the creation of large-scale reuse programs contributed significantly to the field. The US Advanced Research Projects Agency's Software Technology for Adaptable, Reliable Systems initiative,[6] and the many projects sponsored by the European Strategic Programme for Research in Information Technology,[7] especially the Eureka Software Factory, demonstrate the commitment of several governments to promote software reuse. Industry is equally committed, as demonstrated by software factories in Japan (at Hitachi, Toshiba, NEC, Fujitsu, and NTT) and corporate programs in the US (at GTE, AT&T, IBM, and Hewlett-Packard).

Reuse work intensified significantly in the late 1980s. Several advances were made in library systems, classification techniques, the creation and distribution of reusable components, reuse support environments, and corporate reuse programs. In spite of this, a consistent complaint voiced at workshops and conferences was that reuse was not delivering on its promise to significantly increase productivity and quality.

In 1988, Vic Basili broadened the definition of reuse to include the "use of everything associated with a software project, including knowledge."[8] This new perspective has opened doors to research in other disciplines and has contributed to the recognition that the reuse problem is ubiquitous.

Recent work has addressed nontechnical factors: management, economics, culture, and law. We now recognize that these nontechnical problems are as important, and maybe more difficult to solve, than technical problems. Today, our view of reuse attempts to integrate all these factors into the idea of *institutionalized* reuse.

REUSE TAXONOMY

As Table 1 shows, there are at least six perspectives, or facets, from which to view software reuse. The *by-substance* facet defines the essence of the items to be reused; *by-scope* defines the form and extent of reuse; *by-mode* defines how reuse is conducted; *by-technique* defines what approach is used to implement reuse; *by-intention* defines how elements will be reused; and *by-product* defines what work products are reused.

These conceptually independent facets are useful in assessing the state of research, art, and practice and in analyzing problems and issues. All the systems I mention are listed in the box on p. 64.

Idea reuse. This type of reuse involves reusing formal concepts, such as general solutions to a class of problems. It is best exemplified by generic algorithms, such as those developed by Donald Knuth.

The state of algorithm-development research, art, and practice is relatively mature. From the reuse perspective, however, we still must develop and distribute standard catalogs and complement basic catalog information with specific, detailed information on how to reuse the algorithms.

Artifacts reuse. This is where the focus has been: Software factories and corporate programs aimed at repeating Raytheon's experience are examples. The Booch Ada Parts collection and the Generic Reusable Ada Components for Engineers are other good ex-

amples. Early experimentation with parts technology is attributed to the Common Ada Missile Package project, which began in the early 1980s. Today, object-oriented techniques hold the most promise for component reuse — environments like Eiffel and tools like Motif even have integrated object libraries.

Parts-reuse research is now concentrating on quality and reliability, which includes certification. Component adaptability has also become an important issue. Domain-specific collections such as the emerging Computer Software Management and Information Center and the Central Archive for Reusable Defense Software promise to be the new trend in parts reuse.

Procedures reuse. This is one of the most intense research areas right now. Research into process programming and process-centered environments focuses on formalizing and encapsulating software-development procedures. The reuse community is looking into the possibility of creating collections of reusable processes that can be connected to instantiate new and more complex processes. Process reuse is one of the main tasks of the STARS initiative: The STARS Conceptual Framework for Reuse Processes describes a set of reusable processes that can be combined to implement customized reuse programs.

Procedures reuse also means reusing skills and know-how. This area has received significant attention from the expert-systems community, but very little from the reuse community. Project managers tend to reuse skills informally when they reassign personnel, but no formal effort is made to capture and encapsulate knowledge. Domain-analysis researchers are beginning to explore how to reuse expert procedural knowledge. This area has tremendous potential, but it needs more research.

Vertical reuse. Vertical reuse is reuse within the same domain or application area. Its goal is to derive generic models for families of systems that can be used as templates for assembling new systems. The narrower the domain, the bigger the payoff.

The focus of research in vertical reuse has been on domain analysis and domain modeling. Both approaches are concerned with the identification and development of domain models. Several domain-analysis methods have been developed in the last three years. Examples include the Feature-Oriented Domain Analysis, Sandwich, the Synthesis method, and, more recently, the Domain Analysis Guidelines. In addition, several object-oriented methods have been extended to cover domain analysis and modeling.

Although vertical reuse has been practiced informally in most software organizations, the definite trend is toward a more formal, systematic practice. Software factories, for the most part, are aimed at vertical reuse, and organizations with large long-lived projects, such as NASA and the US Department of Defense, are concentrating on vertical reuse. Specialized packages like Motif are examples of successful vertical reuse.

Horizontal reuse. The goal of horizontal reuse is the use of generic parts in different applications. Scientific subroutine libraries, the Unix tools, and the Booch Ada Parts are good examples of horizontal reuse.

Although horizontal-reuse research has been concerned with the packaging and presentation of parts, its current focus is on developing broad-access libraries and library networks. Projects like the Asset Source for Software-Engineering Technology and the Center for Software Reuse Operations are examples of initiatives to develop such repositories. These projects are making significant progress in establishing cataloging and classification standards and interoperability and repository networking. As these initiatives develop, an industry based on software parts for both horizontal and vertical reuse will emerge.

PROCEDURE REUSE IS AN AREA OF INTENSE RESEARCH RIGHT NOW.

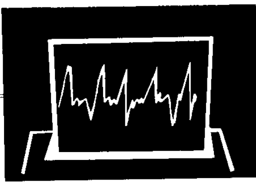
Planned reuse. In planned reuse, the systematic and formal practice of reuse, guidelines and procedures for reuse have been defined, and metrics are being collected to assess reuse performance. It is the norm in software factories.

Planned reuse requires substantial up-front investment and commitment, yet it is difficult to predict the return on investment. Planned reuse requires a significant change in the current practice of software development and demands discipline and compromise from software practitioners.

The recent *Reuse Adoption Guidebook*, which prescribes a step-by-step process to establish a reuse program, is a good example of the state of the art in planned reuse.

Reuse maturity models are at the core of planned reuse, and several now exist. Maturity models define levels that organizations can progress through to achieve formal, systematic reuse. Current reuse maturity models are inspired by the Software Engineering Institute's Capability Maturity Model. The Software Productivity Consortium, for example, has developed a five-level reuse maturity model: Ad-hoc, repeatable, portable, architectural, and systematic.

One problem with planned reuse is the lack of economic models. Managers must know



REPRESENTATIVE REUSE SYSTEMS AND METHODS

Asset Source for Software-Engineering Technology (ASSET): Bldg. 2600, Ste. 2, 2611 Cranberry Sq., Morgantown, WV 26505; (304) 594-3954.
Booch Ada Parts: G. Booch, *Software Components with Ada: Structures, Tools, and Subsystems*, Benjamin/Cummings, Redwood City, Calif., 1987; source code from Wizard Software, 2171 S. Parfet Ct., Lakewood, CO 80215; (303) 986-2405.
Central Archive for Reusable Defense Software (CARDS): USAF Electronics System Center, 1401 Country Club Rd., Fairmont, WV 26554; (304) 367-0421.
Center for Software Reuse Operations: Defense Information Systems Agency/Center for Information Management, 500 N. Washington St., Ste. 101, Falls Church, VA 22046; (703) 536-7485.
Common Ada Missile Package (CAMP): USAF CAMP, AFATL/FXG, Elgin AFB, FL 32542; (904) 882-2961.
Computer Software Management and Information Center (COSMIC): Univ. of Georgia, 382 East Broad St., Athens, GA 30602; (404) 542-3265.
Domain-Analysis Guidelines (draft): DoD Software Reuse Initiative, DISA/CIM, 701 S. Courthouse Rd., Arlington, VA 22204-2199; (703) 285-6589.
Donald Knuth: *The Art of Computer Programming, Vol. 1 (Fundamental Algorithms) 1968,: Vol. 2 (Seminumerical Algorithms) 1969; Vol. 3 (Sorting and Searching), 1973*, Addison-Wesley, Reading, Mass.
Draco: J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.*, Sept. 1984, pp. 564-573.
Domain-Specific Software Architecture (DSSA): Special Report CMU/SEI-92-SR-9, E. Mettala and M.H. Graham, eds., Software Eng. Institute, Pittsburgh, 1992.
Eiffel: Interactive Software Engineering, 270 Stroke Rd., Ste. 7, Goleta, CA 93117; (805) 685-1006.
Feature-Oriented Domain Analysis (FODA): K.C. Kang et al., "Feature-Oriented Domain Analysis," Tech. Report CMU/SEI-90-TR-21, Software Eng. Inst., Pittsburgh, 1990.
Genesis: D. Batory, "Concepts for a Database Compiler," *Proc. Principles of Database Systems Conf.*, ACM, New York, 1988.
Generic Reusable Ada Components for Engineers (GRACE): EVB Software Engineering, 5303 Spectrum Dr., Frederick, MD 21701; (301) 695-6960.
Idea (Intelligent Design Assistant): M. Lubars, "Domain Analysis and Domain Engineering in Idea," in *Domain Analysis and Software Systems Modeling*, R. Prieto-Diaz and G. Arango, eds., IEEE CS Press, Los Alamitos, Calif., 1991, pp. 163-178.
Inquisix: Software Productivity Solutions, 122 Fourth Ave., Indialantic, FL 32903; (407) 984-3370.
Kaptur (Knowledge Acquisition for Preservation of Trade-offs and Underlying Rationales): S. Bailin, "Kaptur: A Tool for the Preservation and Use of Engineering Legacy," tech. report, CTA, Rockville, Md., 1991; (301) 816-1200.
MetaTool: AT&T Bell Laboratories, 20 Shattuck Rd., Andover, MA 01810; (617) 691-3000.
Motif: Open Software Foundation, 11 Cambridge Ctr., Cambridge, MA 02142; (617) 621-8700.
Practitioner: ESPRIT Project 1094, Commission of the European Communities, Directorate Général XIII/A2, 200 rue de la Loi, B-1049 Brussels, Belgium.
Reusable Ada Packages for Information Systems Development (Rapid): SofTech, 460 Tottem Pond Rd., Waltham, MA 02254; (703) 931-7372.
Reuse Based on Object-Oriented Techniques (Reboot): Bull, Rue Jean Jaures, F-78340 Les-Clayes-Sous-Bois, France; (33) 1 30 80 74 48.
Refine: Reasoning Systems, 3260 Hillview Ave., Palo Alto, CA 94304; (415) 494-6201.
Reuse Adoption Guidebook: Tech. Report SPC-92051-CMC, Software Productivity Consortium, Herndon, Va., 1992.
Reuse Library Framework (RLF): Paramax Systems Corp., 12010 Sunrise Valley Dr., Dept. 7720, Reston, VA 22091; (703) 620-7475.
Reuse Library System (RLS): Bldg. 2600, Ste. 2, 2611 Cranberry Sq., Morgantown, WV 26505; (304) 594-3954.
Sandwich Method: R. Prieto-Díaz, "Reuse Library Process Model," Tech. Report CDRL 03041-001, USAF Electronic Systems Center, Hanscom Air Force Base, Mass. 1991.
Software Technology for Adaptable and Reliable Systems (STARS): STARS Technical Center, 801 N. Randolph St., Ste. 400, Arlington, VA 22203; (703) 243-8655.
Synthesis: G. Campbell et al., "Synthesis Guidebook," Tech. Report SPC-91122-MC, Software Productivity Consortium, Herndon, Va., 1991.

the expected benefit, return on investment, initial cost, and performance criteria before they can commit to a full-scale reuse program. Although some economic models have been proposed, they have not been validated. More research and experimentation is definitely needed in economic reuse models.

Ad-hoc reuse. This informal practice, in which components are selected from general libraries, is usually called opportunistic reuse. Ad-hoc reuse is very much the state of the practice: Reuse is conducted at the individual, not the project, level; procedures for reuse don't exist; and the libraries in use contain components not designed for reuse.

The emphasis in ad-hoc reuse has been the development of better and more capable reuse libraries, with friendly interfaces and powerful retrieval mechanisms. The bottleneck, however, has been library population: cataloging and classifying reusable components remains a time-consuming manual task. There has been significant progress in library technology, but more research is needed.

Examples of operational library systems include Reusable Ada Packages for Information Systems Development, which is being used by the US Defense Information Systems Agency/Center for Information Management, and the Reusable Library System, which is being used by ASSET. Several new library systems, like Inquisix, offer advanced features for browsing and retrieval and promise significant improvement in overall reuse. But support for automatic cataloging and classification is still needed.

Compositional reuse. Compositional reuse is the use of existing components as building blocks for new systems. It is based on well-established collections, efficient library systems, and standard interfaces. Although it advocates reuse of all software work products, in practice, compositional reuse has focused mainly on source-code reuse.

Research topics in compositional reuse include component selection and retrieval, component understanding, component adaptation, and component integration. Library systems address the first two, domain analysis the next two, and interface technology addresses integration. Integrated environments are seen more and more as an effective approach to address these issues.

The Unix shell is a typical example of the state of the practice. More recent object-oriented environments like Eiffel reflect the state of the art.

Prototype environments like Reboot (Reuse Based on Object-Oriented Techniques) demonstrate the trend toward formal compositional reuse. Reboot includes associated methods to support reuse by means of object-oriented technology. Other prototype systems include the Reuse Library Framework and Kaptur,

both of which use libraries based on knowledge-representation models like semantic nets.

The state of compositional-reuse research is exemplified by Genesis, a database-management-system generator. Genesis is based on a well-defined conceptual framework of DBMS building blocks that have standard interfaces, module semantics, and potential module interconnections. Genesis users use an open-ended library and a compiler tool to construct DBMSs. The focus of research is on extending and generalizing this approach, and domain analysis is seen as a key element.

Generative reuse. The concept of generative reuse — reuse at the specification level by means of application or code generators — offers the highest potential payoff. However, it is a difficult technology to scale up to industrial production. Systems like Refine and MetaTool demonstrate the potential of this approach and represent the state of the art.

Research in generative reuse has focused on how to formally represent domain-specific specification languages, software processes, and metagenerators. Domain analysis is becoming an important element in deriving vocabularies, architectures, and grammars for specific domains. The trend is toward metagenerators of domain-specific application generators, a goal that would make reuse a natural part of development.

The state of the practice is characterized by domain-specific application generators, many of which are available commercially. Perhaps the best known are Lex and YACC for Unix, which are themselves generators of lexical analyzers and parsers. Research is focusing on extending these approaches to broader domains and to more powerful tools. The state of research is embodied by Draco, which uses domain analysis to develop a formal domain-specific grammar and a set of basic source-to-source transformations. The grammar and transformations are then used with parser generators, pretty-printers, and other generator tools to produce executable code from specifications. There is significant research interest in extending the Draco paradigm.

Black-box reuse. Black-box reuse is the reuse of software components without any modification. Typically, reusable components are packaged and their interactions defined by means of standard interfaces. Although this approach guarantees higher quality and reliability, it is more expensive to create reusable black boxes. Ada and, to some extent, object-oriented programming lend themselves to black-box reuse. The concepts of information hiding and inheritance are key to creating modular, independent components.

One important issue in black-box reuse is verification and certification — how to demonstrate that a component will perform flawlessly under all possible conditions. This problem has drawn significant research, especially formal specifications, which can be used to prove programs correct, and therefore reduce exhaustive testing.

Although black-box reuse has been heralded as the way to create truly reliable systems at low cost, the first collection of certified components has yet to appear. The STARS initiative and several other DoD programs are engaged in creating domain-specific certified components.

White-box reuse. White-box reuse, the reuse of components by modification and adaptation, is by far the most common approach, especially when reuse is ad-hoc. Most reuse programs, including software factories, use white-box reuse. As these programs evolve, reuse by adaptation becomes more formalized. C libraries and the Unix shell are examples of flexible components.

GENERATIVE REUSE HAS THE HIGHEST POTENTIAL PAYOFF, BUT IT IS HARD TO SCALE UP.

The trend in white-box reuse is toward parameterization and built-in adaptability and to expand reuse to higher level products like designs and specifications. Again, domain analysis plays a key role in identifying how families of systems vary. More research in domain analysis would improve white-box reuse.

Reuse products. The by-products facet in Table 1 is a partial but representative list of the kinds of software work products that can be reused.

♦ *Source code.* The state of the practice is to reuse code. Most reuse tools, environments, and methods are aimed at code reuse. As other aspects of reuse evolve, we will see less emphasis on code reuse. Eventually, code will be generated automatically from higher level representations. Developers will not have to deal with source code any more than they deal with assembly code now.

♦ *Designs.* Design reuse offers a higher payoff than code reuse, but it is still an emerging technique. Designs can be partially or indirectly reused through object-oriented methods, but systematic practice is still at the research stage. An example of research prototypes that support reuse at the design level is the Idea environment. Idea supports the reuse of abstract designs represented as design schemas. It uses software specifications to find matching designs from a library.

♦ *Specifications.* When a specification is made available for reuse, it is usually bundled with its respective implementations at the design and code levels. Specification reuse,

which offers the highest payoff of all, is the focus of generative reuse.

♦ *Objects.* Objects are gaining ground as the most reusable work products. There are several methods, tools, and environments for creating objects. The trend in object orientation is toward integrating tools and methods to cover all development phases. Object-oriented methods like Shlaer-Mellor, Rumbaugh, and Booch cover phases from domain analysis to programming and also provide support tools. Object orientation is seen as the technique of the future for reuse. It can be used to support, at least partially, planned, vertical, and compositional reuse.

♦ *Text.* Because all work products, except code, are documents intended for humans, text reuse is becoming increasingly important. Although this aspect is still at the research level, the wealth of knowledge and techniques available from information-retrieval research promises quick progress. The Practitioner project, for example, is developing tools to index, catalog, and manipulate text units. Hypertext is also a key technology. The trend is toward integrating reusable text with all other work products.

♦ *Architectures.* The largest unit of reuse, the architecture is the focus of the Domain-Specific Software Architecture program. Application domains are analyzed to find generic designs that are then used as basic templates for integrating reusable parts or for developing specialized code generators. This research area complements and supports both generative and compositional reuse.

Software reuse has made significant progress in the last 25 years but it is still an elusive technology. Its concepts are simple, yet applying them systematically is a challenge. Three trends have recently emerged that may help us finally solve the reuse problem.

♦ *Domain analysis and domain engineering.* Domain analysis was introduced by Jim Neighbors in 1980 as the activity of "identifying objects and operations of a class of similar systems in a particular problem domain."[9] Domain analysis is the key to systematic, formal, and effective reuse. It is through domain analysis that knowledge is transformed into generic specifications, designs, and architectures. Generic templates will be the basis for creating components that are easy to reuse. Several domain-analysis methods have been proposed in the last few years, and domain-analysis support tools are imminent.

♦ *Process reuse.* The recent interest in characterizing reuse with maturity models and adoption processes is a clear sign of progress toward making reuse a natural part of development. Reuse maturity models provide the necessary framework for the development of tools and methods. Adoption processes define methods and procedures for integrating reuse as standard development practice. Successful integration of reuse guidelines will definitely make a difference: Developers would be eager to adopt reuse, if only they knew how.

♦ *Megaprogramming.* Introduced by Barry Bohem and William Sherlis,[10] megaprogramming is based on domain engineering. Its objective is to integrate current reuse technology, such as library tools, repositories, and techniques, into a systematic framework of industrialized reuse. It has the potential to establish a new development paradigm that is domain-specific, reuse-based, and process-driven. Although megaprogramming resembles the original ideas of McIlroy, the developments of the last 25 years may make it more realizable.

The near future will see significant progress in these areas and hopefully witness the institutionalization of reuse. Reuse, in the end, should come so naturally that we do not have to think about it. ♦

Rubén Prieto-Díaz is president and founder of Reuse, Inc., a software-reuse consulting company. He is contract consultant to Scientific Applications International Corp. for STARS, developed the STARS Reuse Library Process Model, and is participating in the STARS Demonstration Project. He has also supported the ASSET project. He is coeditor of *Software Systems Modeling and Domain Analysis* and the author of several papers on reuse, classification, and domain analysis.

Prieto-Díaz received a BS in aerospace engineering from St. Louis University, an MS in engineering design and economic evaluation and an MS in electrical engineering, both from the University of Colorado at Boulder, and a PhD in computer science from the University of California at Irvine. He is a member of the IEEE Computer Society, IEEE, and ACM.

Address questions about this article to Prieto-Díaz at Reuse, Inc., 12365 Washington Brice Rd., Fairfax, VA 22033; Internet 70410.3014@compuserve.com.

REFERENCES

1. P. Freeman, "Reusable Software Engineering: Concepts and Research Directions." *Proc. Workshop on Reusability in Programming*, Alan Perlis, ed., ITT Programming, Newport, R.I., 1983, pp. 2-16.
2. M.D. McIlroy, "Mass-Produced Software Components," *Software Eng. Concepts and Techniques: 1968 NATO Conf. Software Eng.*, J.M. Buxton, P. Naur, and B. Randell, eds., Petrocelli/Charter, New York, 1969, pp. 88-98.
3. M.A. Cusumano, *Japan's Software Factories*, Oxford University Press, New York, 1991.
4. R.G. Lanergan and C.A. Grasso "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Eng.*, Sept. 1984, pp. 498-501.
5. *Proc. Workshop on Reusability in Programming*, Alan Perlis, ed., ITT Programming, Newport, R.I., 1983.
6. F.W. Martin, "Strategy for a DoD Software Initiative," *Computer*, Mar. 1983, pp. 52-59.
7. Special issue on ESPRIT, *IEEE Software*, Nov. 1989.
8. V.R. Basili and H.D. Rombach, "Toward A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment," Tech. Report CS-TR-2158, CS Dept., Univ. of Maryland, College Park, Md., 1988.
9. J. Neighbors, *Software Construction Using Components*, doctoral dissertation, Information and Computer Science Dept., Univ. of California, Irvine, 1980.
10. B. Bohem and W.L. Scherlis, "Megaprogramming," *Proc. DARPA Software Tech. Conf.* Meridian Corp., Arlington, Va., 1992, pp. 63-82.