Software reuse: From library to factory

by M. L. Griss

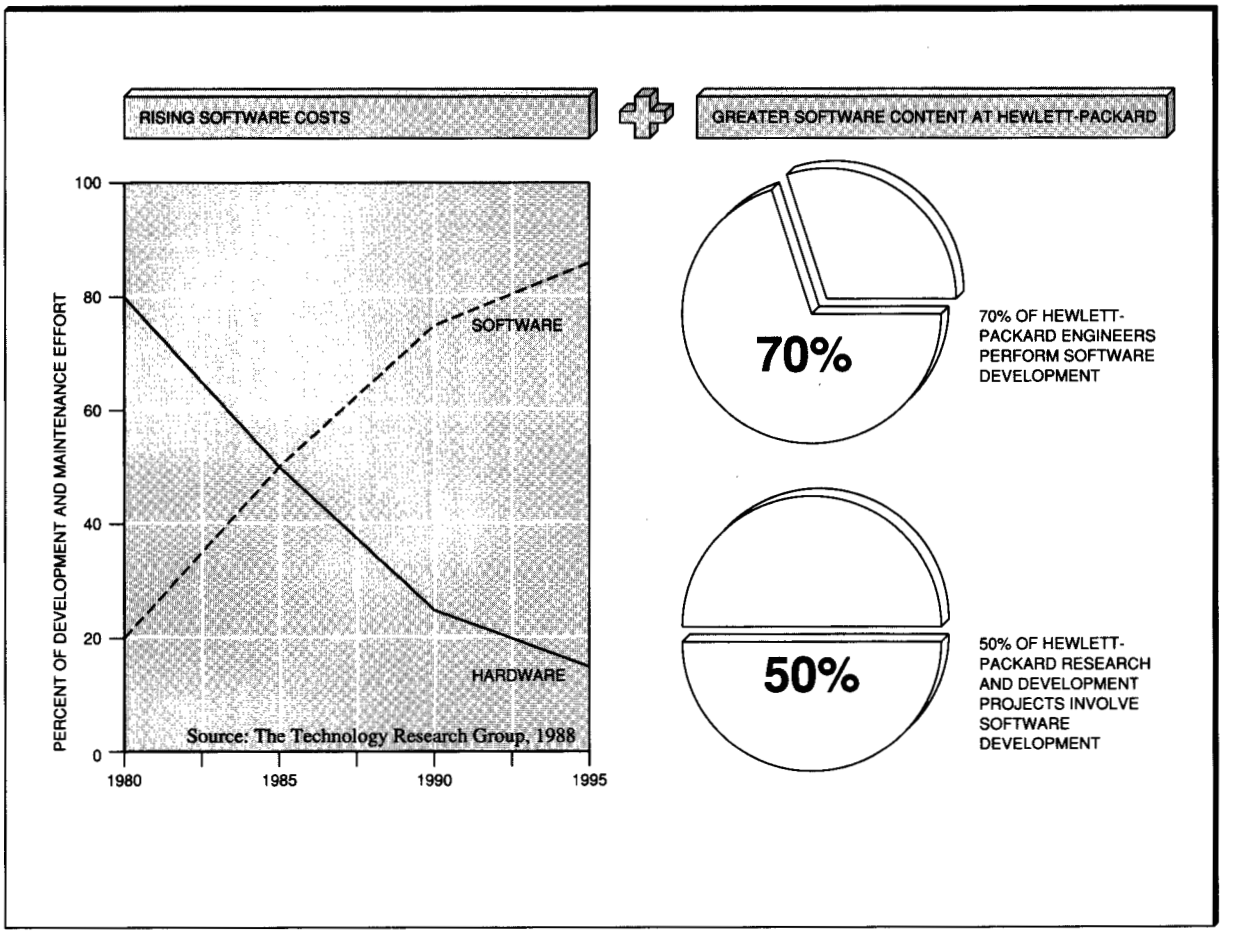
Systematic software reuse is a key business strategy that software managers can employ to dramatically improve their software development processes, to decrease time-to-market and costs, and to improve product quality. Effective reuse requires much more than just code and library technology. We have learned that careful consideration must be given to people, process, and technology. One approach to the systematic integration of these three elements is the concept of the software factory. At Hewlett-Packard Co., we have initiated a multifaceted corporate reuse program to help introduce the best practices of systematic reuse into the company, complemented by multidisciplinary research to investigate and develop better methods for domain-specific, reuse-based software engineering. This essay discusses our experiences. Key aspects include domain-specific kits, business modeling, organization design, and technology infrastructure for a flexible software factory.

The phrase *software crisis* was first used in 1969 to describe the ever-increasing burden and frustration that software development and maintenance have placed on otherwise happy and productive organizations. Since then, managers have been looking for effective strategies to deal with software. Manufacturers of computer systems and instruments, such as Hewlett-Packard Co. and IBM, whose businesses relied mostly on hardware and mechanical engineers, today find that over 70 percent of their research and development engineers are working in the areas of software and firmware. Maintenance and rework account for about 60 to 80 percent of the total software costs. Systems take longer to produce than expected, and software is frequently on the critical path.

Among the many solutions proposed to address this software crisis, the systematic application of software reuse to prototyping, development, and maintenance is one of the most effective ways to significantly improve the software process, shorten time-to-market, improve software quality and application consistency, and reduce development and maintenance costs.[1-3] While many companies are developing proprietary software libraries, software reuse is not yet a major force in most corporate software development. We believe that this is largely because effective reuse depends more on socioeconomic than on technical factors at this time,[4,5] while most users still concentrate on library or language technology.

In this essay I describe how Hewlett-Packard is directing its efforts to better understand these issues and implement solutions to systematically improve and expand its reuse practice. In the following sections I highlight the business issues, the growing need for flexibility, and a more engineered approach to software. I summarize the status and promise of reuse, stressing the integration of improved process, management, and technology. The library metaphor and model, used for many years to guide work in reuse, needs to be replaced by a software engineering model based on kits, factories, manufacturing, and engineering. Software engineers and managers need to change their view of software reuse from that

Figure 1 The software trend



of simply accessing parts in a software library, to that of systematically developing and using well-designed parts following a careful process within a reuse-based software factory.

Business issues facing software managers

The crisis in software is due in part to our very success in designing increasingly complex and powerful computer hardware using integrated circuits and sophisticated computer-aided design and simulation tools, and to our ability to manufacture these tremendously powerful computers with continuously decreasing cost/performance ratios. This has led to the construction of very sophisticated products, incorporating large amounts of software.

There is a growing demand for more complex applications and products that have greater software content, and for more varieties of software than ever before, to serve many more people. New products are appearing ever more rapidly, and in some cases, market windows and product cycles have decreased from several years to only a few months! Unfortunately, software production methods have generally not kept pace (see Figure 1). The increasing size of software systems, more complex standards, and more sophisticated user demands aggravate the situation.

Software managers feel the pressure to simultaneously improve product time-to-market, software quality, and staff productivity. They need to

reduce costs of both development and maintenance. At the same time, they need to maintain or increase the ability to respond effectively and rapidly to changes in markets, requirements, or business cycles. This is sometimes called flexibility or agility, and is believed to be a key to competitiveness in times of change.[6]

Many solutions, but no silver bullet

Many different solutions have been proposed and pursued. They include:

- Higher-level, problem-oriented languages, including so-called fourth-generation languages (4GLs), which both reduce the amount of software written and empower end-users
- Object-oriented analysis, design, and technology that better encapsulate decisions, making it easier for software to evolve and be reused
- Computer-aided software engineering (CASE) tools that provide overall support for the entire life cycle, especially front-end activities such as analysis and design
- Formal methods that ensure accuracy of specifications and interfaces, and therefore improve quality and reduce rework
- Cleanroom, spiral, prototyping, or incremental life cycles that improve the overall software process to reduce development risk and confront requirements change earlier than in the traditional waterfall model
- Inspections, reviews, and structured testing, which ensure and certify quality

Unfortunately, most of these approaches have been pursued with a quick-fix, single-solution attitude. Once the realization "strikes home" that the payoff from any one of the above may be limited and harder to achieve than expected, most organizations abandon that quick fix and move to the next quick fix. They soon discover that there is no single "silver bullet"[7] that will solve the crisis, and most fail to see the need to take the time to integrate each appropriate method into a continuous improvement process.

Systematic software reuse

The idea of systematic reuse (the planned development and widespread use of software components) was first proposed in 1968 by Doug McIlroy.[8] Since then, many attempts at improving the software process by reusing software components have been proposed and tried, with varying degrees of success.

We have come to learn that the key to success with software reuse is a systematic process and a paradigm shift in the way we deal with software development. In the rest of this essay, I describe how software reuse can be made effective by systematically integrating it into an overall software process, and by changing the way we view software reuse and its role in software development.

The dream—a simple concept. The concept is simple. Most statistics on the overall cost and time of software development, and the quality of resulting code, correlate most closely with the amount of new code written. Thus all one has to do to dramatically improve many aspects of the software process is to design and write less new code. At first glance, it appears that all an organization has to do is to collect well-tested software components in a library, and encourage software developers to use these components, rather than to write entirely new ones. In fact, as the amount of a new product made from existing components (the "reuse level") increases, we do observe corresponding improvements in costs, time, and quality. At reuse levels of 80 percent or more, these improvements are dramatic. As I next discuss, it takes more than the parts library alone to achieve these results.

Reuse does work. Several good books, tutorials, and review articles summarize the status of reuse practice and research.[1–3,9,10] Each includes some discussion of the technical, managerial, and organizational implications, as well as important case studies.

Many companies around the world are reporting successful reuse programs.[11,12] These companies include AT&T, The Boeing Co., British-Telecommunications, PLC, Eastman Kodak Company, Ford Aerospace Corporation, Fujitsu Limited, General Dynamics Corp., GTE Corporation, Harris Corporation, Hewlett-Packard, Hitachi, Ltd., Hughes Aircraft Company, IBM, Intermetrics, Inc., McDonnell Douglas Corp., Mentor Graphics Corp., Motorola, Inc., the National Aeronautics and Space Administration (NASA), NEC Corp., Pacific Telecom, Inc., Toshiba Corporation, US West, Inc., and many others. Significant reuse research is underway as part of MCC (Microelectronics and Computer Technology Cor-

poration), SPC (Software Productivity Consortium), SEI (Software Engineering Institute), ESPRIT (European Strategic Programme for Research and Development in Information Technology), STARS, and other programs.

While the books and articles mentioned above cite several benefits, these benefits cannot be calculated and compared easily, since few organizations have kept accurate baseline information, and there are few standard methods of measuring the processes in use today. Many companies are developing proprietary software libraries, but software reuse is not yet a major force in their corporate software development. Nevertheless, it appears that product development costs, factoring in the cost of producing, supporting, and integrating reusable software components, can decrease by a sustainable 10 to 12 percent; defect rates in delivered products can drop drastically to 10 percent of their former levels; and long-term maintenance costs can drop to 20 to 50 percent of their former values when several products share the same, high-quality components.[9,12]

Somewhat different approaches have been followed in Japan, the United States, and Europe. The Japanese approach to reuse has been to concentrate on core functionality, design, productivity, and quality rather than on an ideal feature set, while the approach in the United States has been to concentrate on tools, technology, and feature sets.

The Japanese have been able to produce much higher quality systems at a faster rate.[12] Hitachi reduced the number of late projects from 72 percent in 1970 to 12 percent in 1984, and reduced the number of defects to 13 percent of the 1978 level. Toshiba increased productivity by a factor of 250 percent between 1976 and 1985, and by 1985 had reduced defect rates to 16 to 33 percent of the 1976 level. NEC improved productivity by 126 to 191 percent and reduced defect rates to 33 percent of prior levels. Nippon Telegraph & Telephone Corp. (NTT) has a comprehensive program including a reuse-specific organization, printed catalogs, guidelines, and certification for reuse, leading to reuse levels of 15 percent or more, with several hundred small components.

In the United States, there is significant work at research consortia such as MCC, SPC, SEI, and the Department of Defense-funded STARS program, as well as several companies. IBM has a corporate program, with several reuse support centers, a large parts library, and a multisite Corporate Reuse Council. A key aspect of their program is a formal identification of reuse "champions" and agents.[13,14] Starting in 1988, AT&T developed a domain-specific, large-scale software-bus system for on-line transaction processing and network management.[15] With a support staff of about 30, their reuse program[16] has reduced development costs by about 12 percent and time-to-market from 18–24 months to 6–9 months.

Improvements in cost, time, and quality require more than a parts library.

Most of the European work consists of industrial or industrial-academic consortia. The ESPRIT initiative has funded several industrial-academic collaborations, such as KNOSOS, PRACTITIONER, ITHACA, SCALE, REDO, and REBOOT.[17] REBOOT focuses on object-oriented technology for reuse, providing a reuse-based software engineering environment, a methodology for populating and reusing components, and a base of general-purpose and domain-specific software components. Other efforts include the RACE initiative for telecommunications software, and the EUREKA Software Factory (ESF) initiative and its prototype library management tool. The European Space Agency has a reuse project that emphasizes organizational and contractual issues.[18]

Software reuse at Hewlett-Packard

Hewlett-Packard (HP) has been engaged in software reuse since the early 1980s. Early work involved the development of instrument libraries in BASIC, the construction and use of databases to store and distribute software components, and more recently the use of Objective-C** or C++** to develop class libraries. Several of these librar-

ies have been widely distributed within the company, and some provided to the outside. Today there are many active reuse projects in HP divisions and in HP laboratories. Over the past five years, several HP divisions have been developing and have begun to report on more ambitious reuse programs involving common architectures, components, and libraries for families of related products in a variety of application areas (application "domains"), sometimes spanning several divisions.[19–21] Product areas include: embedded software for instruments and peripherals; network management; and analytical, medical, and manufacturing systems.

Systematic reuse needs to become part of the software process.

HP corporate reuse program. Following an extensive survey in 1989 and 1990 of these ongoing division-sponsored reuse programs and a study of reuse at other companies, a corporate reuse program was created, aimed at making software reuse a more significant and systematic part of the software processes at HP.[22,23] In establishing this program, it was felt that a fairly broad, well-coordinated software reuse program involving management, process, and technology was needed to make significant progress.

The program's goal is to develop, qualify, and promulgate the best practices that can be effective within HP. The program involves a core team of software reuse experts, with additional people working on assignment with several divisional pilot projects. The core team works on developing the following: reuse process, domain analysis, reuse assessments, economic models, coding guidelines, a reuse handbook, reuse education, and consulting to divisional reuse projects. The pilot projects combine the evaluation and refinement of proposed best practices for reuse, and the incremental introduction of new methods into divisional reuse efforts.

The program has been operating since October 1990. It has been responsible for several divisional reuse program assessments, two reuse practitioner's workshops, a draft of the reuse handbook, and several reuse training courses. The core team is actively developing a domain and reusability analysis methodology, and focuses on the methods and processes of managed, systematic, domain-specific software reuse. Unlike some corporations, HP is not building a single corporate-wide centralized reuse library. The diversity of HP's products and divisional software processes requires that divisions create reuse programs and products customized to suit their needs.

Key reuse factors—what we have learned

HP and industry experience have shown that several research and development efforts must be integrated to allow reuse to reach its potential benefit level. Current attempts to introduce libraries and reuse technology have highlighted not only success factors but inhibitors as well.

Reality—people and process get in the way. There are many inhibitors to starting and running an effective reuse program.[24,25] A study of reuse practice at HP and elsewhere has made it strikingly clear that the impediments to improving software reuse are predominantly nontechnical and socioeconomic.[4,5] These include many cultural, organizational, business, or process factors, which can be overcome only through a combination of management, education, process, policy, and incentive initiatives.

Because of early beliefs that reuse required large libraries, many practitioners started their reuse programs by creating classification structures for software components, and collecting as much software (good or bad) as they could. Incentives were offered to contributors. Complex library systems were built to store, manage, and find the components. Researchers have studied alternative classification schemes, prototyped powerful browsers, and automated access systems.

Although the focus was on intriguing technical aspects, much of this work has not produced the desired benefits. Reuse is not as simple as we think.

Libraries of randomly collected code usually only address a small number of a typical developer's needs. Components do not work well together. Developers are reluctant to go looking for components and prefer to write their own, usually claiming (without even checking) that the available parts are wrong, inefficient, or have many defects. Incentives need to be focused on the desired goal. For example, a common incentive is to offer a "reward" for contributing to a library but not for using the library, which can easily increase library size without increasing reuse. This creates a major information problem. People need to know what exists and how to use it.

Systematic reuse will not just "happen." Simply announcing the existence of a library will not cause the paradigm and behavior change desired. Systematic reuse needs to be managed and become an intrinsic part of the software processes that an organization follows. In order to achieve high levels of reuse and so gain the anticipated benefits, pervasive process, organization, and management changes are required. Funding is needed to pay for the ongoing maintenance of components. People need to be trained. Reuse of code alone is not sufficient to produce the large impacts. Since the cost of the coding phase of a project is typically less than 40 percent of the whole, ways to effectively reuse other software work products (such as designs, tests, or documents) are needed to fully exploit the reuse opportunity.

When confronted with their first reuse failure, and suddenly realizing the magnitude of the changes needed to make reuse succeed, organizations need to pursue an incremental improvement process, rather than to abandon the effort. Each reuse organization will encounter varying opportunities and issues peculiar to its situation. For a reuse program to be effective, the specific inhibitors likely to affect it must be identified and overcome in a timely way. A questionnaire-based assessment can be a useful tool to pinpoint key issues. To make it easier to visualize these factors, HP has used a simple assessment framework. It identifies factors such as:

- People—culture, motivation, management, training, skills, and experience
- Process—domain, scope, policies, economics, and standards
- Technology—tools, mechanisms, languages, domain, and architecture

Figure 2 elaborates some of these factors.

Once the dominant inhibitors have been identified, solutions can be proposed and tried. The following are examples of several of the obstacles and possible resolutions.[24,26]

Management. Without the long-term support of senior management and their willingness to make up-front investments, most reuse programs cannot succeed.[27] Such commitment allows projects to work together and balances the short-term needs of individual projects with the longer-term needs of the product portfolio.

The solution may be to pick a suite of projects that already have a supportive senior manager. Sell management on the program, on appropriate time frame expectations, and on the need for commitment by providing them with case studies, cost-benefit analysis, or return-on-investment calculations. Provide contacts with other senior managers who have successful reuse programs in their organizations.

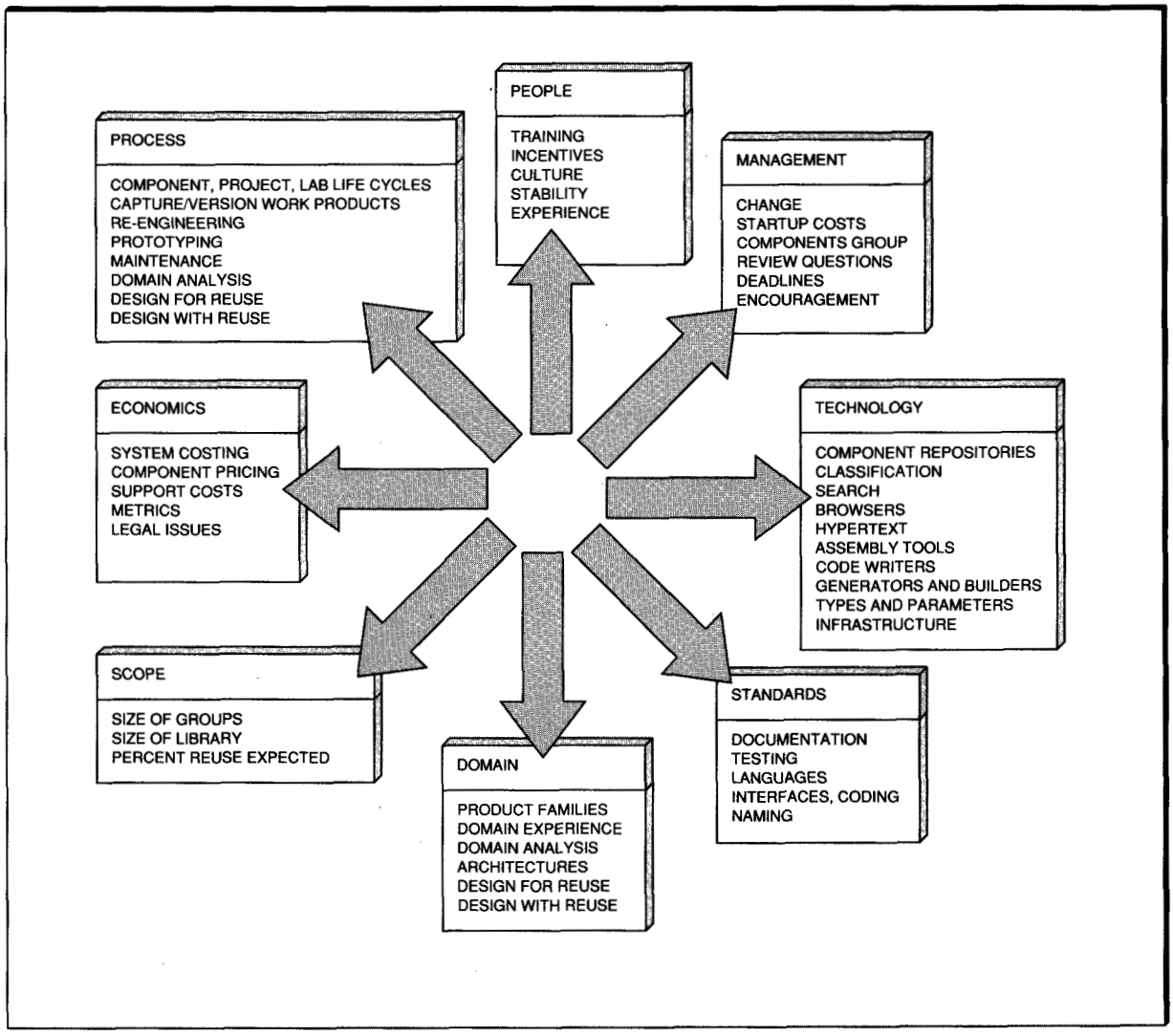
Culture. People do not know how to make use of reuse effectively, or are biased against it through a lack of trust, a "not-invented-here" syndrome, or fear of loss of creativity and independence.

The solution may be to introduce incentives, training, and management backing. Publicize success stories. Try several alternative methods of eradicating the "not-invented-here" syndrome. Build confidence in libraries and support teams.

Organization. Various kinds of institutional barriers make it hard to change financial policies, contracting models, and other legal policies. Evaluation policies such as rewarding individual work and productivity more highly than group work, or complex rules for exchanging and cross-charging for software between divisions or groups can be significant impediments.

The solution may be to create and empower a corporate-wide body (or several, linked, sector- or group-wide bodies) to advocate reuse and make it succeed by changing the reward and funding mechanisms. Establish groups to define and support reusable work products.

Figure 2 Reuse factors



Economics. The funding profile for reuse projects is quite different from conventional software projects. Typically, several years of up-front investment are needed before payoff is realized. Managers are reluctant to make this long-term investment without some guarantee of success.

The solution may be to develop return-on-investment (ROI) models. Describe success stories. Treat reusable work products as assets, requiring appropriate design, maintenance, and enhancement.

Legal issues. In some situations, incorrect contracting mechanisms actively discourage reuse. The lack of contracting mechanisms (i.e., between divisions) makes it hard to create agreements that can be trusted or enforced. Increasing the use of third-party software increases the importance of this issue.

The solution may be to develop (new) contracts, maintenance agreements, and royalty systems. Negotiate for rights to contracted components.

Technical aspects. Arbitrary software work products are typically not very reusable and are often hard to find.

The solution may be to provide guidelines and standards for building, testing, and documenting reusable work products, together with an enforcement mechanism. Sometimes a library with a classification structure, retrieval mechanisms, and certification procedures may be required. Architectural guidelines, documented frameworks, and reuse reviews can help ensure that components are designed to fit. Introduce reuse-oriented inspections to ensure quality and correct component usage.

While there are many opportunities to inject new technology,[28] the nontechnical issues need to be addressed before this technology can be effective. For example, object-oriented technology seems to be a promising vehicle, and recent work in domain analysis, object-oriented methods, library technology, and architectural frameworks has potential to produce a consistent methodology for domain-specific reuse. However, these will not succeed in changing the way we work without a significant effort in the nontechnical areas. For example, most experience suggests that effective systematic reuse will require fairly substantial changes in organization and laboratory-wide software process, which are much harder to introduce than "just" introducing a new method that could be practiced by individuals or (small) independent project teams (such as inspections or configuration management)—and these are not that easy to introduce. People must learn the most effective way to perform cost benefit analysis and evaluation of reuse investments, to set up and use new processes, organizations, and technology, and to provide incentives that encourage appropriate change.

Effective reuse programs. Successful and effective reuse programs begin small, are funded from the start, and have gained experience through pilot projects.

Start small and incrementally. The most effective reuse programs concentrate on the identification and development of a small, high-quality set of needed, useful components, and make sure that the users of these components know about them, know how to use them, and are motivated to do so. Process changes are introduced incrementally, and are adapted as the reuse experience grows. A small, high-quality, well-documented component set is far more effective than a large poor-quality set available on line in a complex library. Once the reuse program is established, the size and scope of the program can be expanded. This means that most engineering and management effort should be (initially) directed at "component engineering"—including design, test, documentation, and support—rather than at the creation of complex library systems and classification schemes. A set of less than 100 components can be handled by simple paper or on-line catalogs.[29] Components are either developed from scratch or re-engineered from existing software. Significant levels of reuse can be achieved in almost any language (COBOL and FORTRAN are common), with very little tool support.

Fund the start-up. Management should fund the creation of components (by "producers"), and also encourage product developers ("consumers") to make use of available reusable components, rather than to write their own. Ensure that producers are aware of, and responsive to, needs of the consumers. Ensure that reuse funding is not diverted. This may be achieved via education, incentives, rewards, edicts, and changes in performance evaluation criteria. Ultimately, economic issues will be significant, but since most reuse programs start with a "tax" funding, rather than "self-funding," there is little industry experience with cost recovery.

Start with a pilot. Early success and quick learning are critical in order to gain and retain management and engineer agreement and support. Risk and cost are also reduced by starting with a small pilot project and proceeding incrementally to expand the scope as the reuse process matures. Early success will facilitate learning about the specific organizational and infrastructure issues that typically impede reuse programs. In some other cases, however, the project cannot succeed without a significant investment and effort to do major system redesign and implementation.

A working group on management and technology transfer at the Fifth Annual Workshop on Software Reuse[30] identified three major stages of reuse adoption, similar to several other reuse adoption or "maturity" models, as follows:

1. *Introduce the commitment to try reuse.* Focus on technology transfer, learning, feasibility study, and starting a pilot project. Expect to encounter and handle resistance. Activities should include determination of the domain bounds, recommendation for the (changes to the) producer and consumer software reuse process, and selection of a domain analysis process and a business model.
2. *Institutionalize the commitment to change and expand the pilot program.* Reuse is integrated into the development cycle. Activities should include facilitation, education, and support, as well as the support and creation of tools and technology. Use appropriate metrics and gather data, record and report on progress and lessons learned, and provide resources. Communicate, motivate people, and maintain project momentum through personal contacts.
3. *Sustain the commitment to improve.* Reuse is integrated into the organization's charter, and technology transfer effort is reduced. Activities include collection of information, provision of support and consultation, assessment and optimization of the reuse producer and consumer processes, and expansion of the scope of the program.

To ensure the success of these stages may require an independent reuse group, distinct component producers, reuse champions, change agents, and corporate sponsors. More complex and detailed adoption and maturity models have been proposed by several reuse researchers[31–33] inspired by the Software Engineering Institute Process Capability Model and Assessment.[34,35] While no common model has yet emerged, all of these models can offer guidance in the design of a reuse program. Each model involves several stages, building on increased learning, experience, confidence, and sophistication. Reuse seems to mature naturally from an *ad hoc* stage to a final managed, systematic, architectural stage.[36]

The payoff where time is of the essence. For many kinds of software development, reducing time-to-market can be even more important than direct cost reduction.[6,37,38] Shorter product cycles can have greater impact on overall profit and competitive advantage than nonreuse development, or even cost-focused reuse-based development. Missing a market window can result in a loss of both market share and a significant portion of the available revenue stream of this product before the next product takes over. A six-month slip in market introduction in a five-year lifetime product can lose more than 27 percent of the potential profit; under these circumstances, spending even 50 percent more during development may be worthwhile. Short product cycles allow rapid learning from the marketplace and allow quicker changes to compensate for competitor moves.

For example, one of Hewlett-Packard's instrument divisions was able to produce the application software for a new product in less than six months. A hardware innovation enabled some new capability, and a quick shift in software priorities allowed a prototype to be deployed at beta-sites in under a month, with final product roll-out four months later. The division general manager asserted that without their prior investment in building up high levels of reuse (nearly 80 percent), they would not have been able to be as responsive.

Improved quality is critical to business success. For products with potentially large sales volumes, high quality increases the acceptance of product upgrades, reduces the cost of on-line service support (for expedient fixes, or "workarounds"), and reduces the consequences of loss of reputation (and market share). For embedded firmware products, recovery from defects shipped in the product can be devastating. Increased cost for field service or product exchange can destroy product profits. For all software and firmware products, improved quality through reuse reduces maintenance costs. HP's peripheral and medical divisions consider reuse as an approach to significantly improve quality and simultaneously reduce the time-to-market. Testing of embedded firmware is critical and time-consuming. A one-month delay can have a dramatic impact on profit. An estimate based on data gathered at one of HP's medical divisions suggests that the (pre-ship) rework costs associated with defects on a typical product can exceed $1 million.[37] These divisions are looking at systematic reuse as a methodology to improve quality and to allow more software developers to be active at the same time in a manageable way. In the case of very large product volumes (such as peripherals or consumer appliances), decreased developer cost has been much less significant than the impact of reduced time-to-market.

Hewlett-Packard Laboratories research

In 1992 HP Laboratories initiated a comprehensive, multidisciplinary software reuse research program to systematically address the plethora of issues described above.[39]

Our research vision is motivated by the notion that components need to fit together well, and that there is a need for a more industrialized approach to quick assembling of new applications from appropriately designed parts using well-designed tools and processes. I am convinced that families of related applications (those in the same application domain space) will increasingly be built from components and assembled within frameworks using glue languages, builders, generators, and other tools. These parts, and the process followed in using them, will be domain-specific to varying degrees.

Using only a library metaphor limits the reuse program results.

I see several implications of this change in the software development paradigm.

- There will be new business opportunities, models, and issues (e.g., how to fund component development, how to package and sell reusable componentry, and how to structure the value chain).
- New organizational structures and specialties will be needed (e.g., distinct component producer and consumer groups, librarians, and domain-analysts). New opportunities for training and management will arise, as well as new cultural and incentive issues.
- New methods will be required to both develop and use the components (e.g., domain analysis, domain engineering, development with reuse, and certification). To support these methods, new tools and technology are possible.

The library metaphor. When we try to describe software reuse, metaphorical terms such as "software library," "software-IC" and "software industrial revolution,"[40] "software-bus,"[41,42] "software Lego,"[43] and "software factory"[44–46] are often used. These suggest and connect with more familiar contexts. While evocative, these carry both useful and confusing (even contradictory) messages.

One attractive metaphor for reuse is the "library" of software "documents," with authors, publications, and librarians. The library metaphor focuses on the management of these documents. This introduces librarians, catalogs, classification schemes, and browsing to the reuse field and has inspired several researchers to investigate various retrieval schemes based on faceted classifications,[47] keywords, or free-text document retrieval.[48] Other researchers are investigating the interoperability and interconnection of local and branch libraries, and the problems of interlibrary loan, copyright, etc. A deeper look at typical industrial reference library systems, with specialist reference librarians, consultants, and acquisition and obsolescence committees, may yield further insight.

While the library metaphor has guided early work in classification and storage systems, we believe that a different metaphor is required for effectively structuring our approach to setting up and running a reuse program. Software modules are not as much like books and articles as we think. Except for encyclopedias, books do not have to "work together," nor are they carefully designed to have compatible, interrelated parts, nor maintained on a careful cycle. Copying software is much easier than copying books. Generally, the resulting cost of buying or borrowing the wrong book is not a major problem. While libraries encourage the acquisition of different books in the same subject area, each displaying an author's individual creativity, rarely are books weighed for which is best, nor is synthesis encouraged—copying and plagiarism are actively discouraged. While software copyright is important, systematic reuse is most often practiced wholly within the confines of one (part of the) company, in which case appropriate software sharing and copying should be encouraged.

While library-based reuse has been fairly successful with stable, well-understood, low-level appli-

cation areas (called "domains" by reuse practitioners) such as user interface libraries, math libraries, and statistical packages, this has not yielded a major change in the way most people develop software.

Software kits, manufacturing, and factories. In the library-centered reuse approach, software code libraries and their usage patterns encourage developers to look for needed parts after most of the design work has been done. By this time, the developer has already made most of the key decisions, and needs to look through large libraries for a part that can be used or adapted, rather than designing the product to available parts.

This shortcoming of the library approach has encouraged us to find and closely examine alternative metaphors for useful insights and direction. A scrutiny of hardware design, product manufacturing, and children's building blocks has provided us with insight. Hardware design and manufacturing involves producing parts to specified tolerances, standards, and processes, and then using design methods that take into account which parts are preferred by a typical organization to enhance manufacturability and reduce overall costs. Similarly, software products need to be designed around the available software components, rather than waiting until the design is done, and then looking (often in vain) for the matching component. This will change the way we design software and software parts, and will require different organizations (called "software factories") to support and enforce these models.

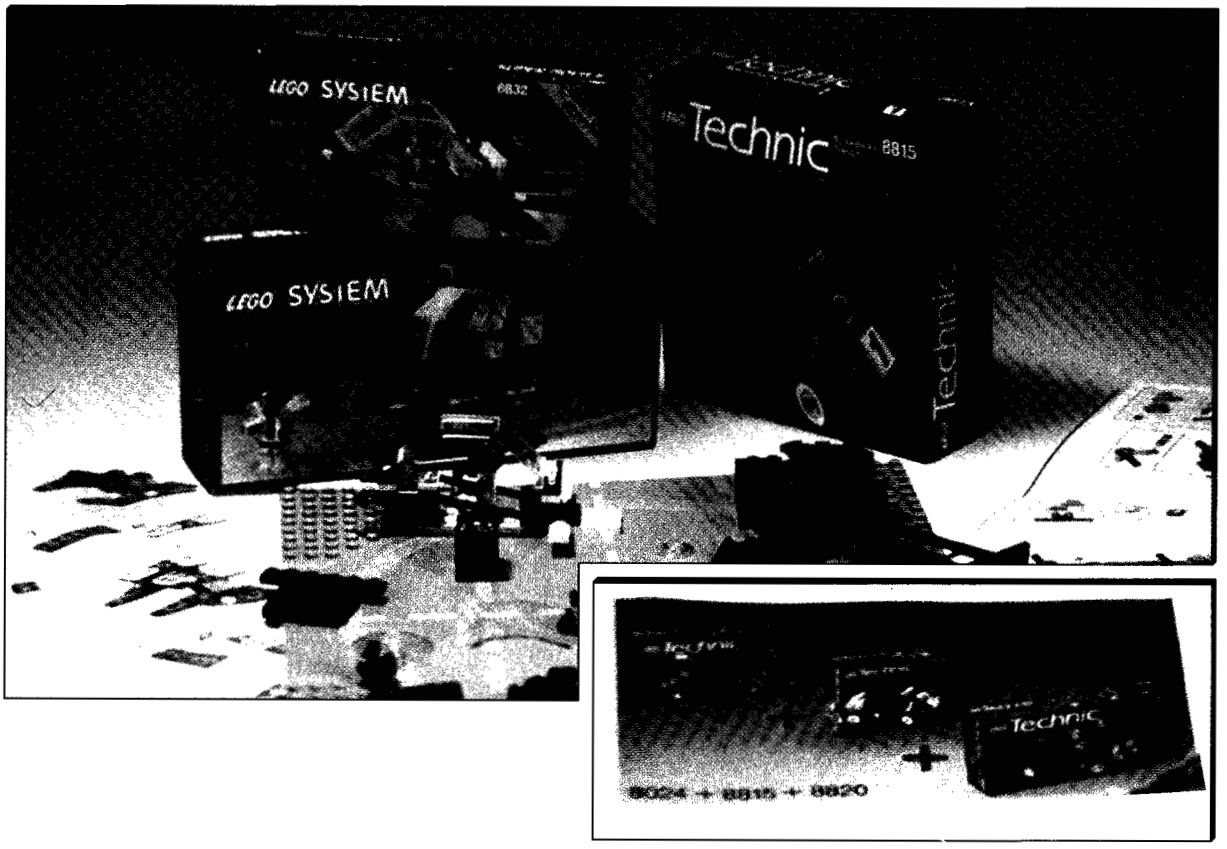
Domain-specific kits. Our vision is that applications will increasingly be built from domain-specific "kits" that consist of several domain-specific parts. These may include: components, frameworks, glue languages, generic applications, tools, environments, and reuse- and domain-oriented processes. We see these kits as a way of packaging compatible domain-specific reusable workproducts, processes, and tools, to cost-effectively cover a space of related applications (a domain). Application developers using a kit to build an application within the domain will find it easier, quicker, and less expensive to use the kit than to build the entire application from the beginning, or to just use random parts from a library.

LEGO** building blocks, a popular children's toy from LEGO Systems, Inc., illustrates well the concept of a kit. The LEGO metaphor has been used by many authors to suggest parts that fit together and (subliminally) exhibit ease of use. Over the years, the LEGO Systems blocks have evolved from a small variety of simple generic parts, to a rich family of kits, composed of a mix of generic and specific parts. Different LEGO kits build spacecraft, farms, castles, and other "domains." Each system comes in carefully-packaged boxes, illustrating the range of related models the kit can build. The boxes are marked with the user "maturity" level for which the kit is most appropriate. Detailed instructions included with the kit specify the process to be followed in using the kit. Application notes are available to indicate how several kits may be combined to build even more complex applications. Some of the series also come with "domain-specific" frameworks (e.g., the space platform, or landing fields). Figure 3 shows some LEGO examples.

While the above example seems perhaps whimsical, it illustrates our view that a "kit" should contain well-designed and packaged compatible reusable work products, tools, and processes to assist in providing more "complete" solutions for application developers. Good kits will substantially reduce the software construction costs and improve the quality, functionality, and interoperability of software in a particular domain.

In the software area, many such kits exist, but they are often seen as toys or curiosities rather than a new way of structuring software. For example, the Pinball Construction Kit and Calculator Construction Kit are well-known end-user kits. HP's HP-VEE and National Instrument's Labview support the construction of instrument systems by engineers, allowing the connection of "virtual" instruments together. Each of these provides an environment in which components (pieces of calculator, pinball game, or instrument) are selected from a palette, and assembled using tools and a visual glue language, to make complete programs that then can be immediately run. Hypercard** for the Apple Macintosh**, and ToolBook** for the IBM-Compatible PCs, provide a similar environment in which components such as buttons, pages, and panels can be glued together and customized to develop a variety of information management and educational applications. These provide a hypertext-based framework, and a programming paradigm based on incrementally adapting a running system (a skeletal

Figure 3 Sample LEGO examples



or generic application), so that prototyping and rapid delivery are enhanced. Additional components (buttons, scripts, widgets, or stacks) can be purchased and added to the available set. Spreadsheet systems are another well-known example. Again, the environment provides a skeletal application that can be adapted into a complete application by a combination of visual programming, macro-language programs, and the loading and modification of complete spreadsheet programs. NeXT Inc. publishes a catalog of objects and kits that can be run in its NextStep** environment.

In a more systematic software process, kits are produced by using methods such as domain analysis, domain engineering, and component engineering, using specialized tools, and perhaps drawing on more generic technologies such as customization kits, UI kits, software-bus frameworks, and architecture and design handbooks.

Our software reuse department in HP Laboratories is researching some of these methods, and is applying them to prototype kits we are currently building. It is our research goal to understand how to build and use kits, how to improve current reuse methods, and how to make their usage a more significant part of the way HP builds software. For example, current domain-analysis methods need to be modified to permit the smooth combination of compositional and generative reuse that are implied by our integrated approach.[49] Our group has built several kits, such as ACEkit for an end-user programmable application construction environment toolkit,[50] and a distributed software-bus kit.[42] We are currently prototyping a new kit in the domain of group-task managers and calendars, using the software-bus as a base kit, with more explicit domain analysis and kit design steps. As the ideas mature, we will work collaboratively with pilot divisional projects to apply the concepts and methods to their situations.

Software preferred parts and group technology. Experience with preferred parts and group technology approaches in manufacturing can also be applied to software. Typical manufacturing organizations expend a significant amount of money[51] to select and qualify vendors and parts. Engineers are then required to design new products using these preferred parts. This simplifies the manufacturing process and amortizes the cost of acquiring the parts. Methods called "design for manufacturability" are common, and engineers are expected to learn these methods and to use the recommended parts. For software, this suggests that we introduce design methods based on preferred software parts, and invest in the qualification and certification of preferred software parts.

To increase reuse and decrease redundancy, an HP software project[52] applied an analysis derived from manufacturing group technology[53] preferred-parts clustering and selection-approach features to do a reuse-oriented early design review. During development of a real-time database, this review resulted in a 25 percent reduction in final code size, and a 40 percent time savings in detailed design, implementation, and unit testing. Overall, the extra time for the reuse review and design change was more than compensated for, resulting in a 25 percent saving in total project cost.

Many computer-aided design (CAD) tools have been augmented to access libraries of preferred parts and give manufacturing cost estimates of using the various parts. This suggests that future reuse-based software environments could help the developer to assemble components into complete systems, as well as comment on the size, cost, quality, and performance implications of selecting a particular part.

Flexible software factory. The other important aspect of our research is a "flexible software factory" in which kits will be constructed and used. Not only must software parts be designed to work together (the notion of a kit) but the component production and product assembly processes must be optimized to decrease redundant engineering and rework. Carefully tuned processes and design guidelines must work together. Attention must be paid to standards for construction, certification, and test.[54]

The flexible software factory provides a framework for analyzing and innovating the way we design, structure, and equip the organizational and technical infrastructure of a software entity to produce, use, and support kits. The term is based on the notion of a software factory, combined with the idea of flexible manufacturing systems.

While the explicit title "software factory" was first applied by the Software Development Corporation (SDC) in the United States to develop a more engineered approach to software, it was in Japan that the experience of actually setting up and optimizing hardware manufacturing organizations was applied to extend these early ideas of a software factory.

Over the years, the Japanese manufacturing industries have introduced a series of innovations that have increased the flexibility, focus, and agility of factories. These include the focused factory and the flexible factory.[6] The key idea is to reduce unnecessary steps, reduce the number of different parts, and optimize the materials flow (such as just-in-time processing) to reduce time-to-market and the time it takes to set up a new product line.

The initial factories and assembly lines were set up to achieve economies of scale to make vast numbers of essentially identical products, using carefully tuned processes, specialized tools, and optimized material flow, justifying huge capital investment in a rather rigid plant and process. As described by M. Cusumano,[44] software factories are more appropriately aimed at economies of scope, obtained by sharing standard parts and processes across a variety of related products.

Toshiba, Hitachi, and others[12,44] invested large amounts of money to train, equip, and measure organizations of many thousands of programmers in order to systematically improve large-scale software development. They created a series of focused software organizations that employ high levels of software reuse in limited classes of applications (called application domains) to more quickly produce related applications. Simple repeatable and standardized processes are used and optimized by continuously gathered metrics.

While the early notions of a software factory are also somewhat rigid,[44,45,55] software construction needs flexible, adaptable organizations akin to

those introduced in flexible manufacturing to achieve economies of scope.

Different authors use the term software factory to focus on different aspects of a more industrialized production of software. Some focus on the organizational aspects, emphasizing distinct specialties and roles. Others focus on the use of rigorous, high-maturity software processes and metrics. Some focus on reuse, while others emphasize automated toolsets and powerful environments. Finally, others address the training, learning, and flexibility aspects. As indicated by Cusumano,[44] these are all facets of applying factory-like ideas to software.

V. Basili uses the term "experience factory" or "knowledge factory"[56] to emphasize the flexible information-processing and learning aspects of a reuse-based organization. M. Patterson[57] applies the manufacturing or factory metaphor to product development as well as software development, mapping the processing of raw materials and manufactured goods into the gathering, processing, and use of information. M. Simos discusses organizational design and learning in his papers on the Organon.[58,59] W. Scacchi emphasizes process support, reuse, and tools in his University of Southern California system factory.[60] The term "new software factory"[61] has been used by Lockheed Corp. to refer to a domain-specific technical infrastructure for domain-specific software engineering, which partially explores organizational roles. J. Eng describes a Bellcore "workstation software factory."[46]

For us at HP Laboratories, the key is to integrate several factory and manufacturing derived concepts together to understand how the cultural, organizational, people, management, technical, and process issues play together as an organization builds software by creating, using, supporting, and evolving domain-specific kits. Our sense of "flexibility" is coupled with the notion that different processes and styles of kits are most appropriate for different styles of software business and organizational experience. To do this, our multidisciplinary research department includes experts in organizational design, anthropology, software process modeling, and business as well as traditional software technologists and engineers.[39] Figure 4 illustrates the concepts of the HP domain-specific kits and the flexible software factory.

Accordingly, our idea is to perform a business-specific and domain-specific design or redesign by following these steps:

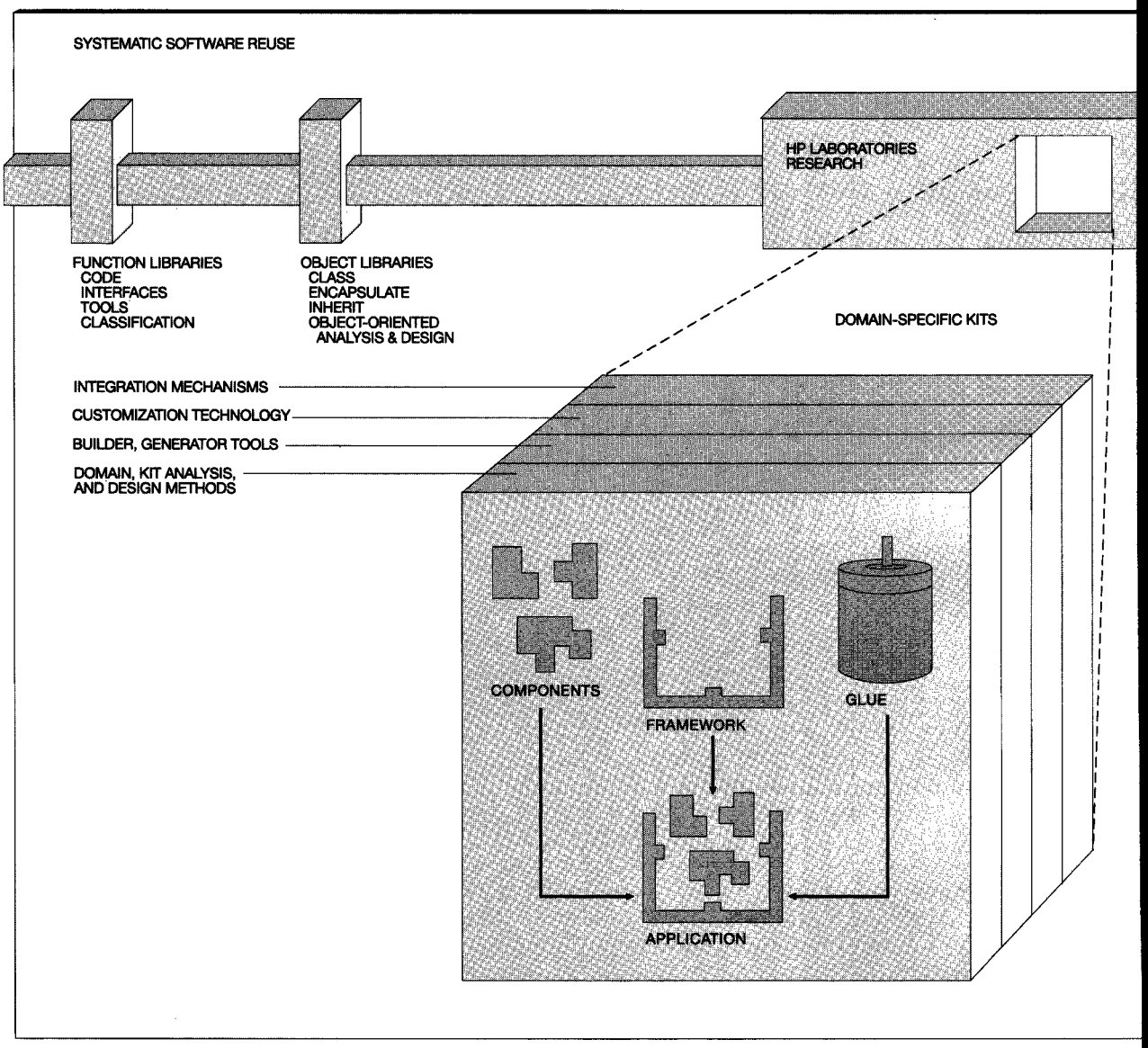
- *Model the business* (business analysis) to identify the key issues this variant of the reuse-based business may address, as well as influence and determine the domain choice.
- *Identify key software reuse process elements* (technical analysis) and relate these to marketing, product definition, and software development processes.
- *Define the new reuse-oriented organization* (social analysis) starting from the reuse process elements, a set of design guidelines and elements adapted from the experience and redesign of knowledge work organizations, reuse organizations, and the software processes used by software entities.
- *Provide the factory with a supporting technical infrastructure* (tools and environments) appropriately customized from a "kit" of flexible software factory support components and framework.

Identifying the key software reuse process elements and adapting the reuse organization to optimize these is similar in concept to binding the organization roles in Basili's reuse reference architecture,[56] the process selection phase of B. Boehm's spiral model,[62] or the use of "core process" based organization and business redesign. Tools and environments in the flexible software factory will be built on an open information management and computer-supported collaborative work substrate (built on a distributed software bus,[42] a multiuser hypertext,[63] and an open software tool integration framework such as HP SoftBench**.[64] This will support both traditional software development and CASE tools, and also new software reuse and process enactment tools (such as Matisse,[65] HP SynerVision**, domain-analysis tools, browsers, builders, and generators). This framework will permit customizing both the software process and tool set to support the specific kind of reuse being done.

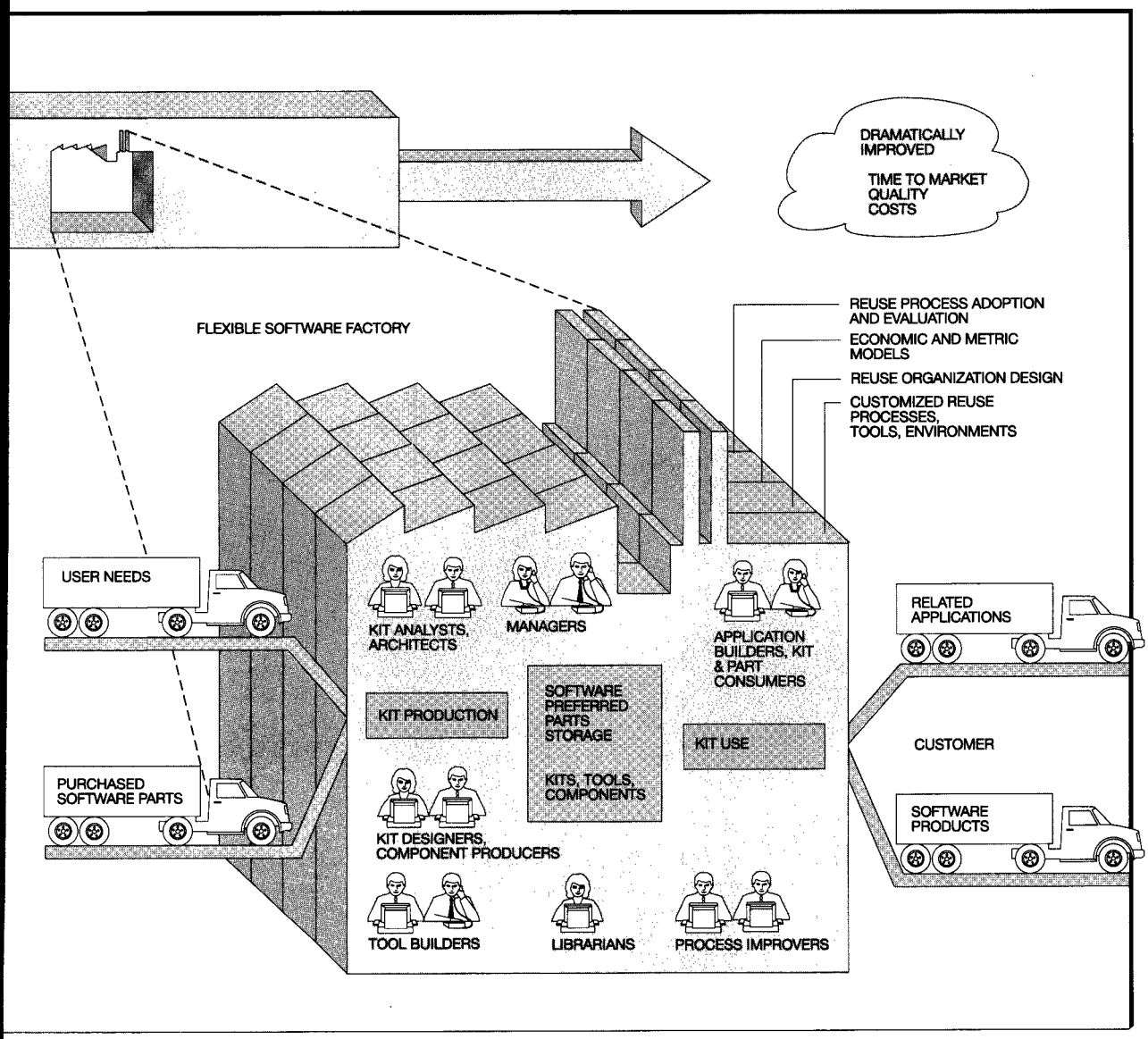
Conclusion

Software reuse can work and can produce satisfying savings in cost and time, even though traditional problems in measuring software productivity make it difficult to quantify the savings. Software reuse, however, is not a quick-fix silver

Figure 4 A flexible software factory



bullet. To make software reuse work takes more effort than the simple and familiar software library metaphor would suggest. Inhibitors to successful systematic reuse can be overcome by improved processes, careful management, and well-designed and packaged reusable software. New methods and technologies can yield highly reusable domain-specific kits, comprising reusable components, frameworks, and glue languages. New processes and organizations can produce flexible and effective software factories. These approaches offer great promise for further gains, and a more systematic attack on the software problem.



Acknowledgments

I received many useful suggestions from Patricia Collins, Danielle Fafchamps, Mehdi Jazayeri, Reed Letsinger, James Navarro, Ruth Malan, Marv Patterson, Chuck Untulis, and Kevin Wentzel. I am particularly grateful for the efforts and dedication of my research assistant, Marty Wosser. She read and corrected numerous drafts of the essay, found key references, investigated HP preferred parts processes, and helped prepare the figures.

**Trademark or registered trademark of Stepstone, Inc., AT&T, LEGO Systems, Inc., Apple Computer, Inc., Asymetrix Corp., NeXT, Inc., or Hewlett-Packard Co.

Cited references and notes

1. T. Biggerstaff and A. Perlis, *Software Reusability*, Volumes 1 & 2, ACM Press, New York (1989).
2. W. Tracz, *IEEE Tutorial on Software Reuse: Emerging Technology*, IEEE Computer Society Press, IEEE Catalog Number EH0278-2 (1988).
3. B. Barnes and T. B. Bollinger, "Making Reuse Cost Effective," *Software* **8**, No. 1, 13–24 (January 1991).
4. B. J. Cox, "Planning the Software Industrial Revolution," *Software* **7**, No. 6, 25–33 (November 1990).
5. G. Aharonian, "Starting a Software Reuse Effort at Your Company," (distributed at) NASA Workshop, *Towards a National Software Exchange* (April 1991); available through Source Translation and Optimization, P.O. Box 404, Belmont, MA 02178.
6. G. Stalk, Jr., "Time—The Next Source of Competitive Advantage," *Harvard Business Review* **66**, No. 4, 41–51 (Jul–Aug 1988).
7. F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* **20**, No. 4, 10–19 (April 1987).
8. D. McIlroy, "Mass-produced Software Components," *Proceedings of Software Engineering Concepts and Techniques*, 1968 Nato Conference on Software Engineering, J. M. Buxton, P. Naur, and B. Randell, Editors (January 1969), pp. 138–155; available through Petrocelli/Charter, New York (1969).
9. *Software Reusability*, W. Schaefer, R. Prieto-Diaz, and M. Matsumoto, Editors, Ellis Horwood, Chichester, England (1993).
10. J. W. Hooper and R. Chester, *Software Reuse—Guidelines and Methods*, Plenum Press, New York (1991).
11. P. Walton, "Software Reuse: Management Issues," *Proceedings of the First International Workshop on Software Reusability*, R. Prieto-Diaz et al., Editors (July 1991), pp. 100–107; available through SWT Memo Internes Memorandum des Lehrstuhls, Software-Technologie, Prof. Dr. Herbert Weber, Fachbereich Informatik, Universität Dortmund, Postfach 500 500, D-4600 Dortmund 50, Germany.
12. R. H. Yacobellis, *A White Paper on U.S. vs. Japan Software Engineering*, Technical Report, Motorola, Austin, TX 78767 (January 1990).
13. J. R. Tirso, "The IBM Reuse Program," *Proceedings of the Fourth Annual Workshop on Software Reuse*, L. Latour, Editor, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1991), pp. 1–5.
14. J. R. Tirso, "Championing the Cause: Making Reuse Stick," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. Tirso 1–6.
15. K. J. Anderson, R. P. Beck, and T. E. Buonanno, "Reuse of Software Modules," *AT&T Technical Journal* **67**, No. 4, 71–76 (July 1988).
16. R. P. Beck, S. R. Desai, D. R. Ryan, R. W. Tower, D. Q. Vroom, and L. M. Wood, "Architectures for Large Scale Reuse," *AT&T Technical Journal* **71**, No. 6, 34–45 (November/December 1992).
17. J. Faget and J. Morel, "The REBOOT Approach to the Concept of a Reusable Component," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. Faget 1–7.
18. J. Favaro, "Measuring the Cost of Reusable Ada Components," *Proceedings of First Symposium on Ada in Aerospace*, Barcelona (December 1990).
19. R. Martin, G. Jackoway, and C. Ranganathan, "Software Reuse Across Continents," *Proceedings of the Fourth Annual Workshop on Software Reuse*, L. Latour, Editor, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1991), pp. 1–7.
20. G. Mayobre, "Using Code Reusability Analysis to Identify Reusable Components for the Software Related to an Application Domain," *Proceedings of the Fourth Annual Workshop on Software Reuse*, L. Latour, Editor, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1991), pp. 1–14.
21. A. Nishimoto, "Evolution of a Reuse Program in a Maintenance Environment," *Proceedings of the Second Irvine Software Symposium—ISS'92*, R. W. Selby, Editor, Irvine Research Unit in Software, Department of Information and Computer Science, University of California, Irvine, CA 92717 (March 1992), pp. 89–108.
22. P. Collins, "Considering Corporate Culture in Institutionalizing Reuse," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. Collins 1–4.
23. K. Harris, "Increasing Reusability Through Architectural Design," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. 1–5.
24. W. B. Frakes, "An Empirical Framework for Software Reuse Research," *Third Annual Workshop: Methods & Tools for Reuse*, CASE Center, Syracuse University, Syracuse, NY 13244-4100 (June 1990), pp. 1–5.
25. W. Tracz, "Software Reuse: Motivators and Inhibitors," *Digest of Papers*, Computer Society International Conference (COMPCON) Spring '87; Thirty-Second IEEE Computer Society International Conference on Intellectual Leverage, IEEE Cat. No. 87CH2409-1 (February 1987), pp. 358–363.
26. M. L. Griss, J. Favaro, and P. Walton, *Managerial and Organizational Issues—Starting and Running a Software Reuse Program*, Ellis Horwood, Chichester, England (1993), Chapter 3, pp. 51–78.
27. It is also true that many successful reuse programs have a "corporate angel" who funds and nurtures the program through its early years.
28. W. Tracz, "Software Reuse Technical Opportunities," *Proceedings of the DARBP Computer Science Conference*, Los Angeles, CA (April 1992).
29. S. Isoda, "An Experience of Software Reuse Activities," *Proceedings of the First International Workshop on Software Reusability*, R. Prieto-Diaz et al., Editors (July 1991), pp. 79–85; available through SWT Memo Internes Memorandum des Lehrstuhls, Software-Technologie, Prof. Dr. Herbert Weber, Fachbereich Informatik, Universität Dortmund, Postfach 500 500, D-4600 Dortmund 50, Germany.
30. *Proceedings of the Fifth Annual Workshop on Software*

Reuse, M. L. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992).
31. M. J. Davis, "Stars Reuse Maturity Model: Guidelines for Reuse Strategy Formulation," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. M 1–6.
32. R. Prieto-Diaz, "Making Software Reuse Work: An Implementation Model," *Software Engineering Notes* **16**, No. 3, 61–68 ACM Press, New Yoık (July 1991).
33. P. Koltun and A. Hudson, "A Reuse Maturity Model," *Proceedings of the Fourth Annual Workshop on Software Reuse*, L. Latour, Editor, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1991), pp. 1–4.
34. W. S. Humphrey and W. L. Sweet, *A Method for Assessing the Software Engineering Capability of Contractors*, Technical Report CMU/SEI-87-TR-23 ESD/TR-87-186, Carnegie Mellon University, September 1987; available through Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.
35. W. S. Humphrey, "Characterizing the Software Process: A Maturity Framework," *Software* **5**, No. 2, 73–79 (1988).
36. "Architectural" implies domain analysis, architectures, frameworks, and well-designed, compatible building blocks. "Systematic" or "managed" implies well-defined and intentional reuse processes and carefully-structured organizations.
37. W. T. Ward, "Calculating the Real Cost of Software Defects," *Hewlett-Packard Journal* **42**, No. 4, 55–58, 3200 Hillview Ave., Palo Alto, CA 94304 (October 1991).
38. P. G. Smith and D. G. Reinertsen, *Developing Products in Half the Time*, Van Nostrand Reinhold, New York (1991).
39. M. L. Griss, "A Multi-Disciplinary Software Reuse Research Program," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. Griss 1–8.
40. B. J. Cox, "There Is a Silver Bullet," *BYTE* **15**, 209–218 (October 1990).
41. J. Purtilo, *The Polylith Software Bus*, Technical Report CSD 2469, Computer Science Department, Institute for Computer Studies, University of Maryland, College Park, MD 20742 (1990).
42. B. W. Beach, M. L. Griss, and K. D. Wentzel, "Bus-based Kits for Reusable Software," *Proceedings of Irvine Systems Symposium (ISS)'92*, University of California at Irvine, Irvine, CA (March 1992), pp. 19–28.
43. E. Corcoran, "Soft Lego: How Software Designers Hope to Make Programs Reusable," *Scientific American*, 145–146 (January 1993).
44. M. A. Cusumano, *Japan's Software Factories*, Oxford University Press, New York (1991).
45. W. S. Humphrey, "Software and the Factory Paradigm," *Software Engineering Journal* **6**, No. 5, 370–376 (September 1991).
46. J. Eng, "Implementing a Software Factory at Bellcore," *Proceedings of the Fifth Annual Workshop on Software Reuse*, M. Griss and L. Latour, Editors, Department of Computer Science, University of Maine, Orono, ME 04469 (November 1992), pp. Eng 1–8.
47. R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Proceedings of the Twelfth International Conference on Software Engineering*, IEEE (March 1990), pp. 300–304.
48. W. B. Frakes and B. A. Nejmeh, "An Information System for Software Reuse," *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse 1987*, IEEE Computer Society Press, Holmdel, NJ (1988), pp. 142–151.
49. M. Griss and W. Tracz, "Workshop on Software Reuse," *ACM Software Engineering Notes* **18**, No. 2, 74–85 (April 1993).
50. J. Johnson, B. Nardi, C. L. Zarmer, and J. R. Miller, "Ace: Building Interactive Graphical Applications," *Communications of the ACM* **36**, No. 4, 41–55 (April 1993).
51. HP has a group of over 40 people within its corporate procurement organization whose job it is to assign part numbers and maintain a database of over 100 000 parts ranging from resistors to ICs, gears, and complete subsystems.
52. F. Fatehi, "Group Technology + Design for Reusability = On-Time Quality Software," [HP] R&D Network, 18–21 (January 1990).
53. Group technology is a manufacturing methodology that groups parts into families by coding prominent characteristics such as shape, materials, and manufacturing steps.
54. Cox, in Reference 4, suggests that software "gauges" can be used to help certify that components meet their specifications within acceptable tolerances.
55. For example, Kodak refers to their reuse endeavor by the rather static term "software warehouse."
56. V. R. Basili, G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology* **1**, No. 1, 53–80 (January 1992).
57. M. L. Patterson, *Accelerating Innovation*, Van Nostrand Reinhold Co. Inc., New York (1993).
58. M. Simos, "The Domain Life Cycle: Steps Toward a Unified Paradigm for Software Reusability," *IEEE Tutorial on Software Reuse: Emerging Technology*, Will Tracz, Editor, IEEE Computer Society Press, 1730 Massachusetts Ave., N.W., Washington, DC, 20036-1903 (1988).
59. M. A. Simos, "Software Reuse and Organizational Development," *Proceedings of the First International Workshop on Software Reusability*, R. Prieto-Diaz et al., Editors (July 1991), pp. 36–41; available through SWT Memo Internes Memorandum des Lehrstuhls, Software-Technologie, Prof. Dr. Herbert Weber, Fachbereich Informatik, Universität Dortmund, Postfach 500 500, D-4600 Dortmund 50, Germany.
60. W. Scacchi, *The Software Infrastructure for a Distributed System Factory*, Technical Report, University of Southern California, Los Angeles, CA 90089-0782 (June 1990); revised version appeared in *Software Engineering Journal* **6**, No. 5, 355–369 (September 1991).
61. W. Mark, *Ceres: A New Software Factory*, Technical Report, Lockheed, Palo Alto, CA (April 1992).
62. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer* **21**, No. 5, 61–72 (May 1988).
63. M. Creech, D. Freeze, and M. L. Griss, "Using Hypertext in Selecting Reusable Software Components," *Proceedings of Hypertext'91* (Software and Systems Laboratory, Palo Alto, CA), ACM, New York (December

1991), pp. 25–38. (See also HP Laboratory Technical Report SSL-91-59.)
64. M. R. Cagan, "The HP Softbench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal* **10**, 36–47, 3200 Hillview Avenue, Palo Alto, CA 94304 (June 1990).
65. P. Garg, T. Pham, B. Beach, A. Desphande, W. Fong, A. Ishizaki, and K. Wentzel, *Matisse: A Knowledge-based Team Programming Environment*, Technical Report HPL-92-104, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304 (August 1992); to appear in *International Journal of Software Engineering* (1994).

Accepted for publication March 31, 1993.

Martin L. Griss *Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94306 (electronic mail: griss@hpl.hp.com).* Dr. Griss is Technical Director for Software Engineering and a laboratory scientist/engineer in the Computer Research Center at Hewlett-Packard Laboratories, Palo Alto. He leads research on software reuse, software factories, software-bus frameworks, and hypertext-based reuse tools. He works closely with HP corporate engineering to systematically introduce software reuse into HP's software development processes. He was previously director of HP's Software Technology Laboratory, researching expert systems, object-oriented databases, programming technology, human-computer interaction, and distributed computing. Before that, he was Associate Professor of Computer Science at the University of Utah, working on computer algebra and portable LISP systems (PSL). He received a Ph.D. in physics from the University of Illinois in 1971.

Reprint Order No. G321-5524.