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, we have initiated a multi-faceted Corporate Reuse Program to help introduce systematic reuse best-practices into the company, complemented by multi-disciplinary research to investigate and develop better methods for domain-specific, reuse-based software engineering. Key aspects include domain-specific kits, business modeling, organization design, and technology infrastructure for a flexible software factory.
1 Introduction

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 and IBM, whose business relied mostly on hardware and mechanical engineers, today find that over 70% of their R&D engineers are engaged in software and firmware. Maintenance and rework account for about 60% to 80% of 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[BP89, Tra88, BB91]. 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 technical factors at this time[Cox90a, Aha91], while most users still concentrate on library or language technology.

In this paper 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 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.

2 Business Issues Facing Software Managers

This intolerable software situation is due in part to our very success in designing increasingly complex and powerful computer hardware using integrated circuits and sophisticated CAD 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 (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 (TTM), 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 some-
times called flexibility or agility, and is believed to be a key to competitiveness in times of change[JR88].

3 Many Solutions, But No Silver Bullet

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

- Higher-level, problem oriented languages, including so-called 4th 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 software easier to evolve and reuse.
- CASE tools which provide overall support for the entire lifecycle, especially upstream activities such as analysis and design.
- Formal methods which ensure accuracy of specifications and interfaces, and therefore improve quality and reduce rework.
- Cleanroom, spiral, prototyping, or incremental lifecycles which 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” [Bro87] that will solve the crisis, but most fail to see the need to take the time to integrate each appropriate method into a continuous improvement process.

4 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 [McI69]. 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 paper, I will describe how software reuse can be made effective, by systematically integrating it into an overall software process, and changing the way we view software reuse and its role in software development.

4.1 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, rather than writing new ones from scratch. 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% or more, these improvements are dramatic! As we will see below, it takes more than the parts library alone to achieve these results.

4.2 Reuse Does Work

There are several good books, tutorials and review articles that summarize the status of reuse practice and research [SPDM93, BB91, HC91, BP89, Tra88]. 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 [Wal91, Yac90]. These companies include IBM, AT&T, Hewlett-Packard, Toshiba, Motorola, GTE, NEC, Fujitsu, Hitachi, British-Telecom, Mentor Graphics, US West, Intermetrics, PacTel, Boeing, Ford Aerospace, General Dynamics, McDonnell Douglas, NASA, Hughes, Harris, Kodak, and many others. Significant reuse research is underway as part of MCC, SPC, SEI, ESPRIT, STARS and other programs.

While the references cite several benefits, these benefits cannot be calculated and compared easily, since few organizations have kept accurate baseline information nor use the same
methods of calculation. 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 10-12%; defect rates in delivered products can drop drastically to 10% of their former levels; and long term maintenance costs can drop to 20%-50% of their former values when several products share the same, high-quality components [Yac90, SPDM93].

Somewhat different approaches have been followed in Japan, the US and Europe. The Japanese approach to reuse has been to concentrate on core functionality, design, productivity and quality rather than ideal feature set, while the US approach 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 [Yac90]. Hitachi reduced the number of late projects from 72% in 1970 to 12% in 1984, and reduced the number of defects to 13% of the 1978 level. Toshiba increased productivity by a factor of 250% between 1976 and 1985, and by 1985 had reduced defect rates to 16%-33% of the 1976 level. NEC improved productivity by 126%-191% and reduced defect rates to 33% of prior levels. NTT has a comprehensive program including a reuse-specific organization, printed catalogs, guidelines and certification for reuse, leading to reuse levels of 15% or more, with several hundred small components.

In the United States, there is significant work at research consortia such as the MCC, SPC, SEI and the DoD-funded STARS program, as well as several companies. IBM has a Corporate Program, with several Reuse Support Centers, a large parts library, and a multi-site Corporate Reuse Council. A key aspect of their program is a formal identification of reuse champions and agents [Tir91, Tir92]. Starting in 1988, AT&T developed a domain-specific, large-scale software-bus system for OLTP and Network management [ABB88]. With a support staff of about 30 [BDR+92], their reuse program has reduced development costs by about 12% and time-to-market from 18-24 months to 6-9 months.

Most of the European work consists of industrial or industrial-academic consortia. The ESPRIT initiative has funded several industrial-academic collaborations, such as KNOSOS, FRACTITIONER, ITHACA, SCALE, REDO and REBOOT [FM92]. 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 initiative Software Factory (ESF) and its prototype library management tool. The European Space Agency has a reuse project that emphasizes organizational and contractual issues [Fav90].

5 Software Reuse at Hewlett-Packard

Hewlett-Packard has been involved with software reuse since the early 1980's. 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 (such as UI, and instrument sub-systems). Several of these libraries have been widely distributed within the company, and some provided to the

1There are few standard methods of measuring software work products and processes in widespread use.
outside. Today there are many active reuse projects in HP divisions and in HP Laboratories.

Over the past 5 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 [MJR91, May91, Nis92]. Product areas include: embedded software for instruments and peripherals; network management, and chemical, medical, and manufacturing systems. These divisions are involved with various aspects of the reuse process, setting up libraries, developing architectures and designs, doing domain-analysis, establishing reuse goals (such as deposition ratios, or reuse levels), and establishing and running reuse councils, and component and generator production projects.

5.1 HP Corporate Reuse Program

Following an extensive survey in 1989/90 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 HP’s software processes [CoI92, Har91]. 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 best-practices that can be effective within HP. The program consists of 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 pilots combine the evaluation and refinement of proposed reuse best-practice, and the incremental introduction of new methods into divisional reuse efforts.

The program has been operating since October 1990. It has assessed several divisional reuse programs, held two Reuse Practitioner’s Workshops, released a draft of the reuse handbook, and offered several reuse training courses. The core team is actively developing a Domain and Reusability Analysis methodology, and focusing on the methods and processes of managed, systematic, domain-specific software reuse. Unlike some Corporate reuse programs, 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.

6 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.

6.1 Reality - People and process get in the way

There are many inhibitors to starting and running an effective reuse program [Fra90, Tra87]. A study of reuse practice at HP and elsewhere has made it strikingly clear that the impediments to improving software reuse are predominantly non-technical and socioeconomic
[Cox90a, Aha91]. 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.

Everyone has been having a great time focusing on some intriguing technical aspect. But, 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 amount 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, or inefficient, or have lots of 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.

What a mess. Yet another “silver bullet” gone astray.

Systematic reuse will not just “happen.” Simply announcing the existence of a library will not cause the paradigm and behavior change we want. 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 coding phase of a project is typically less than 40% of the whole, ways to effectively reuse other software workproducts (such as designs, tests, 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 giving up. 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, experience
- PROCESS - domain, scope, policies, economics, standards
- TECHNOLOGY - tools, mechanisms, languages, domain, architecture
Using a simple 5 point scale, each factor is displayed on one axis of a Kiviat diagram, as in Figure 2. Successful reuse programs will have activities or plans along many of the dimensions. They should appear with an “appropriate” shape, perhaps balanced, rounded, or lobed in a certain way. The evolution of the program can be seen in the changing shape. Figure 3 displays a diagram showing the evolution of reuse at one of HP’s networking divisions [May91].

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 [Fra90, GFW93]:

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

*Solutions:* 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.

**Cultural:** People do not know how to reuse effectively, or are biased against it through a lack of trust, “not-invented-here” syndrome (NIH), fear of loss of creativity and independence.

---

\[2\text{Many successful reuse programs have a “corporate angel,” who funds and nurtures the program through its early years.}\]
Figure 3: Division reuse factors

**Solutions:** Introduce incentives, training and management backing. Publicize success stories. Try several alternative methods of eradicating the NIH syndrome. Build confidence in libraries and support teams.

**Organizational:** Various kinds of institutional barriers will make it hard to change financial policies, contracting models and other legal policies. Personal 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.

**Solutions:** Create and empower a corporate-wide body (or several, linked, sector- or group-wide bodies) to advocate reuse and make it succeed, by changing reward and funding mechanisms. Establish groups to define and support reusable workproducts.

**Economic:** 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.

**Solutions:** Develop Return-on-Investment (ROI) models. Describe success stories. Treat reusable workproducts as assets, requiring appropriate design, maintenance and enhancement.

**Legal:** In some situations, incorrect contracting mechanisms actively discourage reuse. Lack of contracting mechanisms (i.e. between divisions), make it hard to create agreements that can be trusted or enforced. Increasing use of third party software increases importance of this issue.
Solutions: Develop (new) contracts, maintenance agreements, and royalty systems. Negotiate for rights to contracted components.

Technical: Arbitrary software workproducts are typically not very reusable and are often hard to find.

Solutions: Provide guidelines and standards for building, testing and documenting reusable workproducts, together with an enforcement mechanism. Sometimes a library with 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[Tra92], the non-technical issues need to be addressed before this technology can be effective. For example, object-oriented technology seems to be a promising vehicle for realizing the dream, 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 non-technical areas. For example, most experience suggests that effective systematic reuse will require fairly substantial changes in organization and lab-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 of performing cost/benefit analysis and evaluating reuse investments, in setting up and using new processes, organizations and technology, and in providing incentives that encourage appropriate change.

6.2 Effective Reuse Programs

Start small and incremental. The most effective reuse programs concentrate on identifying and developing a small, high-quality set of needed, useful components, and on ensuring 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, adapting as the reuse experience grows. A small, high-quality, well-documented component set is far more effective than a large poor-quality set available online in a complex library. Once the reuse program is established, the size and scope of the program can be expanded. This means that most engineer and management effort should be (initially) directed at “component engineering,” including design, testing, documenting and support, rather than at creating complex library systems and classification schemes. A set of less than 100 components can be handled by simple paper or online catalogs[Is091]. 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 startup. Management should fund the creation of components (by “producers”), and also encourage product developers (“consumers”) to make use of available reusable components, rather than writing their own. Ensure that producers are aware of, and responsive to, needs of 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 buy-in 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.

The “Management and Technology Transfer” working group at the 5th Annual Workshop on Institutionalizing Reuse (WISR'92)[GL92] identified three major stages of reuse adoption, similar to several other reuse adoption or “maturity” models:

1. Introduce - Commitment to try reuse. Focus on technology transfer, learning, feasibility study, and starting a pilot project. Expect to encounter and handle resistance. Activities include determining the domain bounds, recommending (changes to the) producer and consumer software reuse process and selecting a domain analysis process, if any and a business model.

2. Institutionalize - Commitment to change and expand the pilot program. Reuse is integrated into the development cycle. Activities include facilitation, education and support, as well as 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 - Commitment to sustain and improve. Reuse is integrated into the organization’s charter, and technology transfer effort is reduced. Activities include collecting information, providing support and consulting, assessing and optimizing the reuse producer and consumer processes and expanding 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[Dav92, PD91, KH91] inspired by the SEI3 Process Capability Model and Assessment[Hea87]. 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.4

6.3 The Payoff - Time is of the essence

For many kinds of software development, reducing time-to-market can be even more important than direct cost reduction[War91, Jr88, SR91]. Shorter product cycles can have the greater impact on overall profit and competitive advantage than non-reuse 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 6-month slip in market introduction in a 5-year

---

3The Software Engineering Institute.
4Architectural 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.
lifetime product can lose more than 27% of the potential profit; under these circumstances, spending even 50% more during development may be worth it (see figure 4). Short product cycles allow rapid learning from the market place, 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 6 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 4 months later. The division general manager asserted that without their prior investment in building up high levels of reuse (near 80%), they would not have been able to be as responsive. A group building onlinetransaction processing and network management systems at AT&T has experienced similar systematic and dramatic decrease in time-to-market from 18 to 6-9 months on systems as large as 500K lines[BDROUT92]. They experienced reuse levels on the order of 90%.

Improved quality is critical to business success. For products with potentially large sales volumes, high quality reduces the threat of product upgrades, the cost of on-line service support (for workarounds) and 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 improving quality and simultaneously reducing 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 $1M[War91]. These divisions are looking at systematic reuse as a methodology to improve quality, and to al-
low 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.

7 Hewlett-Packard Laboratories Research

In 1992 HP Laboratories initiated a comprehensive, multi-disciplinary software reuse research program to systematically address the plethora of issues described above [Gri92].

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 quickly assembling new applications from appropriately designed parts using well designed tools and processes. I am convinced that families of related applications\(^5\) will increasingly be built from components, 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.

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/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 componentry, e. g. domain analysis, domain engineering, developing with reuse, and certification. To support these methods, new tools and technology are possible.

7.1 The Library Metaphor

When trying to describe software reuse, metaphorical terms such as "Software Library," "Software-IC" and "Software Industrial Revolution" [Cox90b], "Software-Bus"[Pur90, BGW92], "Software LEGO"[Cor93] and "Software Factory"[Cus91, Hum91, Eng92] 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, publication, and librarians. The "library" metaphor focuses on the management of these documents. This introduces to the reuse field librarians, catalogues, classification schemes, browsing, etc. This has inspired several reuse researchers to investigate various retrieval schemes based on faceted classifications[PD90], keywords, or free-text document retrieval [FN88]. Other researchers investigate the interoperability and interconnection of local and branch libraries, and the problems of inter-library loan, copyright, etc. A deeper look at typical industrial reference library systems, with specialist reference librarians, consultants and acquisition/obsolescence committees may yield further insight.

\(^5\)These applications are related within a space called an "application domain."
While the library metaphor has guided early work in classification and storage systems, it is the wrong metaphor 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 encyclopaedias, 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 downstream cost of buying/borrowing the wrong book is not a major problem. While libraries encourage the acquisitions 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 application 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.

7.2 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 have 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.

7.3 Software LEGO - Domain-specific kits

Our vision is that applications will increasingly be built from domain-specific "kits" consisting 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 cheaper to use the kit than to build from scratch or to just use random parts from a library.
LEGO Systems building blocks, a popular children's toy, illustrates well the concept of a kit. The LEGO metaphor has been used by many authors to suggest parts that fit together and (subliminally) 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 Systems address space craft, farms, castles and other “domains.” Each system comes in carefully-packaged boxes, illustrating the range of related models this box 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). See Figure 5.

While the above example seems perhaps whimsical, it illustrates our view that a “kit” should design and package compatible reusable workproducts, tools and processes to provide 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 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 which then can be immediately run. Hypercard for the MAC, and ToolBook for the PC, 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 or generic application), so that prototyping and rapid delivery are enhanced. Additional components (buttons, scripts, widgets, 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 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, architecture and design handbooks, etc. See Figure 6.

Our Software Reuse Department in HP Laboratories is researching some of these methods, and is applying them to prototype kits we are 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. Our group has built several kits, such as ACEkit for an end-user programmable application construction environment toolkit[JNZM93], and a distributed software-bus kit[BGW92]. We are 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.

7.4 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 to select and qualify vendors and parts. Engineers are then required to design new products using these preferred parts. This will simplify the manufacturing process, and amortize 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[Fat90] applied an analysis derived from manufacturing “group technology” preferred parts clustering and selection approaches features to do a reuse-oriented early design review. During development of a real-time database, this review resulted in a 25% reduction in final code size, and a 40% time savings in detailed design, implementation and unit testing. Overall, the extra time for the

---

*See the summary of the WISR’92 Domain Analysis working group[GT93].

7HP 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 IC’s, gears, and complete sub-systems.

*Group technology is a manufacturing methodology that groups parts into families by coding prominent characteristics such as shape, materials, manufacturing steps, etc.
reuse review and design change was more than compensated for, resulting in a 25% saving in total project cost.

Many 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.

7.5 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.9 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. Our term plays on the notion of software factory, combined with the idea of flexible manufacturing systems.

While the explicit title “software factory” was first applied by SDC10 in the US to develop a

---

9 Cox suggests that software “gauges” can be used to help certify that components meet their specifications to within acceptable tolerances (Cox90a).

10 The Systems Development Corporation.
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 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 [Jr88]. The key idea is to reduce unnecessary steps, reduce the number of different parts, and optimize the materials flow (such as "just-in-time", JIT) 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 rather rigid plant and process. As described by Cusumano [Cus91], 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 [Yac90, Cus91] invested large amounts of money in training, equipping and measuring organizations of many thousands of programmers, 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 Software Factory are also somewhat rigid [Cus91, Hum91], 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, stressing distinct specialties and roles. Others focus on the use of rigorous, high-maturity software processes and metrics. Some focus on reuse, while others stress automated toolsets, and powerful environments. Finally, others address the training, learning and flexibility aspects. As indicated by Cusumano [Cus91], these are all facets of applying factory-like ideas to software.

Basili uses the term "Experience Factory" or "Knowledge Factory" [BCC92] to stress the flexible information processing and learning aspects of a reuse-based organization. Patterson [Pat93] applies the manufacturing/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. Simos discusses organizational design and learning in his papers on the Organon [Sim88, Sim91]. Scacchi stresses process support, reuse and tools in his USC System Factory [Sca90]. The term "New Software Factory" [Mar92] has been used by Lockheed to refer to a domain-specific technical infrastructure for domain-specific software engineering, which partially explores organizational roles. Eng describes a Bellcore "Workstation Software Factory" [Eng92].

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,

---

11 For example, Kodak refers to their reuse endeavor by the rather static term, "Software Warehouse"  
12 We are partnering with a group in HP that has experience in manufacturing line redesign.
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 multi-disciplinary re-
search department includes experts in organizational design, anthropology, software process
modeling and business as well as traditional software technologists and engineers[Gri92].

Accordingly, our idea is to perform a business- and domain- specific “reuse organization
(re-)design” by doing the following steps:

**Business Analysis** - model the business to identify the key issues this variant of the reuse-
based business may address (as well as influence/scoping the domain choice).

**Technical Analysis** - identify key “software reuse process” elements, and relate these to
marketing, product definition and software development processes.

**Social Analysis** - define the new reuse-oriented organization, starting from the reuse pro-
cess 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.

**Tools and Environments** - provide the factory with a supporting technical infrastructure,
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[BCC92], the process selection phase of Boehm’s spiral model[Boe88],
or the use of “core process” based organization and business redesign. Tools and envi-
ronments in the flexible software factory will be built on an open information manage-
ment and computer-supported collaborative work substrate (built on a distributed software
bus[BGW92], a multi-user hypertext[CFG91]) and an open software tool integration frame-
work such as HP SoftBench[Cag90]. This will support both traditional software develop-
ment and CASE tools, and also new software reuse and process enactment tools (such as
Matisse[GPB+93], domain-analysis tools, browsers, builders and generators). This frame-
work will permit customizing both the software process and toolset to support the specific
kind of reuse being done.

### 8 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 bullet. To make software reuse
work takes more effort than the simple and familiar software library metaphor would sug-
gest. 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 produced flexible and
effective software factories. These approaches offer great promise for further gains, and a
more systematic attack on the software problem.
9 Acknowledgment

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 paper, found key references, investigated HP preferred parts processes, and helped prepare the figures.
10 References


11 Biography

Martin L. 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 an 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.