Improve Software Quality by Reusing Knowledge and Experience

Victor R. Basili • Gianluigi Caldiera

THE APPROACHES FOR IMPROVING QUALITY IN MANUFACTURING PROCESSES DON'T WORK ESPECIALLY WELL FOR SOFTWARE DEVELOPMENT. THE AUTHORS provide a quality improvement paradigm for the software industry that builds on manufacturing models but focuses on reused learning and experience by establishing "experience factories." Their iterative process enables an organization to acquire core competencies to support its strategic capabilities.

Victor R. Basili is a professor and Gianluigi Caldiera is a research associate at the Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland.

The quality movement that has had such a dramatic impact on all industrial sectors has finally reached the systems and software industry. Although some of the concepts of quality management originally developed for other products can be applied to software, as a product that is developed and not produced, it requires a special approach. In this paper, we introduce a quality paradigm specifically tailored to the systems and software industry. We discuss the reuse of knowledge, products, and experience as a feasible solution to the problem of developing higher-quality systems at a lower cost. In other words, how can an organization build models or package them so that it can reuse them in other projects?

Companies often achieve quality improvement by defining and developing an appropriate set of strategic capabilities and supporting core competencies. We propose a quality improvement paradigm (QIP) for developing core competencies. This process must be supported by a goal-oriented approach to measurement and control, and an organizational infrastructure that we call an experience factory.

In this paper, we introduce the major concepts of our proposed approach, discuss their relationship with other approaches in the industry, and present an example of an organization that successfully applied those concepts.

Why Is Software Development Different?

Software is present in almost every activity and institution of our society. Our dependence on software becomes evident when software problems — system shutdowns, new product delays, and assorted glitches — become newspaper headlines. The business community is aware of these problems but does not truly understand their causes. Such misunderstanding extends to the software business community itself, especially when it deals with the philosophies of quality improvement.

> Problems often arise when companies try to transfer the quality lessons learned in the manufacturing process to the software development process.

Problems often arise when companies try to transfer the quality lessons learned in the manufacturing process to the software development process. Quite often, manufacturers develop quality models by collecting great amounts of data from work locations where the same function is repeated over and over. In such a context, statistical quality control can be accomplished based on numerous repetitions of the manufacturing process. Because software is developed once, this type of control is impossible. Software development models, therefore, cannot be built the same way as manufacturing models, with their dependence on lessons learned from massive repetitions of the same process. Software models provide something less definitive — the ability to learn from other software development projects. To accomplish this learning, we have to distinguish what is different about these projects.

A company can manage the quality of a software system in two ways. First, it can improve the effectiveness of the software development process by reducing the amount of rework and by reusing software artifacts across segments of a project or different projects. Second, it can develop and implement plans for controlled, sustained, and continuous improvement based on facts and data.

A major problem with software engineering is that data regarding a system's quality can be observed and measured only when the system is implemented. Unfortunately, at that stage, correcting a design defect requires the expensive redesign of sometimes large, complex components. To prevent expensive defects from occurring in the final product, quality management must focus on the early stages of the engineering process. At those early stages, however, the process is less defined and controllable with quantitative data. Therefore, software engineering projects do not regularly collect data and build models based on them.

There are many successful software projects from a quality point of view. Quality management's goal is to repeat this success in other projects by transferring the knowledge and experience at the roots of that success to the rest of the organization. A software organization that manages quality should have a corporate infrastructure that links together and transcends the single projects by capitalizing on successes and learning from failures.

Organizations need to have a strategic approach to software quality management as a part of a corporate strategy for software, aimed at pursuing and improving quality on an organizational level. There is no solution that can be mechanically transferred and applied to every organization (the famous "silver bullet"). Every organization can use our proposed approach, however, after appropriate customization, to improve software quality in a controllable way.

The Problem of Software Quality

How does a company improve quality in a *development* environment instead of a *production* environment? The key is to build or package models so that they are reusable by other projects in the organization — that is, to reuse knowledge and experience.

In many disciplines, quality issues are well understood. Because of the relative newness of the software business, definitions or trade-offs aren't clear. Software users often can't articulate what qualities they really want. Do they care about reliability, user-friendliness, or ease of modification? Software doesn't really break in the normal sense, but it has to evolve. Today's system won't satisfy the user three years from now because there are constantly changing expectations.

Because software is a new field, and good, sound models are hard to build, companies have not built models to reason about what things are, how they work, and what they should look like. Quality isn't defined so that both the developer and the user can understand it and communicate it.

Of the approaches to software quality available, there are various paradigms, mostly from manufacturing. Some organizations apply an improvement process to their software processes based on the Shewart-Deming cycle.[1] This four-stage approach provides a way to manage change throughout the production process by analyzing the change's impact on the data derived from the process:

1. Plan — define quality improvement goals and targets and determine methods for reaching those goals; prepare an implementation plan.
2. Do — execute the implementation plan and collect data.
3. Check — verify the improved performance using the data collected from the process and take corrective actions when needed.
4. Act — standardize the improvements and install them into the process.

Some organizations use the total quality management (TQM) approach, which is derived from the Shewart-Deming method and applied to all the company's business processes.[2] Another approach is benchmarking, in which organizations model their improvement on an external scale that represents the best practices in quality. The goals of the improvement program are, in this case, not internally generated but suggested by the best practices.

The software industry has used these approaches — and variations on them — with mixed outcome. The major problem is that these approaches do not deal specifically with the nature of a software product. Or if they do, they assume a consistent picture of a good software product or process. This is not adequate because, to be really effective, a software quality program should deal with the nature of the software business itself. There is no such thing as an explicit, consistent picture of a good software product.

Our approach reflects an attempt to learn from the successes of the different paradigms and to avoid problems when they are applied to software environments.

Table 1 Traditional and Expanded Focus of Software Development

Traditional Focus	Expanded Focus
Delivering specific products and services	Developing capabilities
Decomposing a complex problem into simpler ones	Unifying different solutions into more general ones
Designing and implementing	Analyzing and synthesizing
Detailing	Abstracting from detail
Validating and verifying	Experimenting

We rely on the lean enterprise concept by concentrating production and resources on value-added activities that represent an organization's critical business processes.[3]

Toward a Mature Software Organization

Successful management strategies of the past ten years all call for long-term investments and top management sponsorship.[4] They advocate establishing a permanent structure to develop and support the reuse of strategic capabilities. This strategy is new for the software industry, which is predominantly driven by its business units and therefore has little ability to capitalize on experiences and capabilities.

Companies that develop software have sought to apply recent management strategies in the following ways:

1. The company must understand the software process and product.
2. The company must define its business needs and its concept of process and product quality.
3. The company must evaluate every aspect of the business process, including previous successes and failures.
4. The company must collect and use information for project control.
5. Each project should provide information that allows the company to have a formal quality improvement program in place, i.e., it should be able to control its processes, tailor them to individual project needs, and learn from its own experiences.
6. Competencies must be built in critical areas of the business by packaging and reusing clusters of experience relevant to the company's business.

Software companies need to expand their focus on a new set of problems and the techniques for solving them. Unfortunately, a software project is traditionally based on a case-by-case, problem-solving approach; the development of strategic capabilities is based instead on experience reuse and organizational sharing. (Table 1 outlines the traditional focus of software development and problem solving, along with the expanded focus.)

A Strategy for Improvement

At the center of an improvement strategy is the need for reusable experience. Next we present the framework of our strategy through a process we call the quality improvement paradigm. We discuss an approach to quality improvement based on the development of strategic capabilities, on a control tool (the goal-oriented approach to measurement that addresses the support of the improvement process with quantitative information), and on an organizational tool (an infrastructure aimed at capitalization and reuse of software experience and products).[5]

Are there any practical models a company can use to develop a strategy with the new focus? Later we illustrate with an example of a practical model, which we chose because it is a unique blend of an organizational strategy aimed at continuous improvement, a data-based approach to decision making, and an experimental paradigm, along with many years of continuous operation and data collection.

The Quality Improvement Paradigm

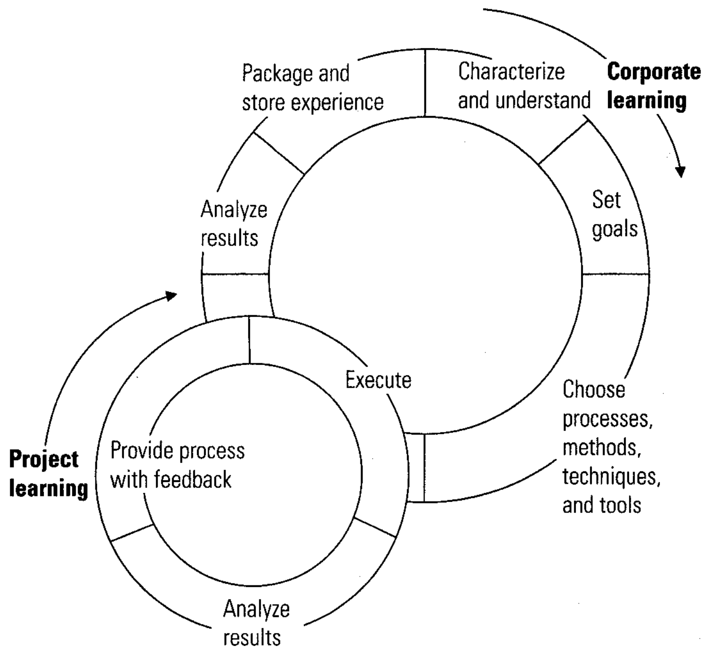
A common problem of software development companies is that they don't think software is their business. They think they are building "telephone systems" or "switching systems" when they are really building telephony software and switching software. They have little understanding of strategic capabilities and core competencies.

In the software business, companies determine strategic capabilities by knowing whether they can reuse architectures and designs, what functionality their product has, and how to estimate the cost of adding new features or changing existing ones. Strategic capabilities are always supported by core competencies — technologies tailored to the specific needs of the organization in performing business processes.

The goal of the process we present here is the acquisition of core competencies that support strategic capabilities. The organization must own, control, and properly maintain competencies as state of the art and know how to tailor them to the characteristics of specific projects and business units.

The quality improvement process occurs in six steps (see Figure 1). By *characterizing*, a company builds models of the current environment. Next it *sets goals* for what it wants to achieve for the next product and learn about the business. To satisfy the goals relative to the current environment, it *chooses processes, methods, techniques, and tools*, tailors them to fit the problem, and *executes* them. During

Figure 1 The Quality Improvement Paradigm


Package and store experience
Characterize and understand
Corporate learning
Set goals
Analyze results
Execute
Choose processes, methods, techniques, and tools
Project learning
Provide process with feedback
Analyze results

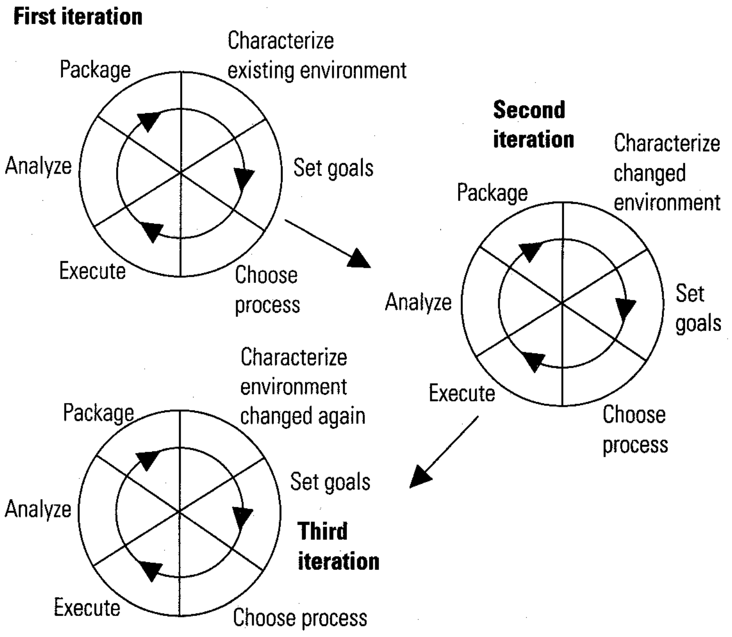
execution, it analyzes the intermediate results and asks if it is satisfying the goals and using appropriate processes. This feedback loop is project learning. Finally, the company *analyzes* what happened and learns from it. Then it stores and propagates the knowledge, i.e., *packaging*.

Each cycle results in better models in terms of characterization of the software business, a better articulation of goals, and a better understanding of the relationship between processes and their effects. Each time through the loop is a corporate learning event.

The quality improvement paradigm implements two major cycles:

- The control cycle is the feedback to the project during the execution phase. It provides analytic information about project performance at intermediate stages of development by comparing project data with the nominal range for similar projects. This information is used to prevent and solve problems, monitor and support the project, and realign the process with the goals.
- The capitalization cycle is the feedback to the organization. Its purpose is to understand what happened, by capturing experience and devising ways to transfer that experience across application domains and to accumulate reusable experience in the form of software artifacts that are applicable to other projects and are improved based on the analysis.

Figure 2 The Quality Improvement Paradigm as an Iterative Process



An organization's use of the quality improvement paradigm is an iterative process that repeatedly characterizes the environment, sets appropriate goals, and chooses the process for achieving those goals. It then proceeds with the execution and analytical phases. At each iteration, it redefines and improves characteristics and goals (see Figure 2).

Goal-Oriented Measurement

The goal/question/metric (GQM) approach provides a method to identify and control key business processes in a measurable way.[6] A company can use it to define metrics during the software project, process, and product so the resulting metrics are tailored to the organization and its goals and reflect the quality values of different viewpoints (developers, users, operators, and so on).

A GQM model is a hierarchy starting with a goal (specifying purpose of measurement, object to measure, issue to measure, and viewpoint from which to take the measurement). Suppose a company wants to improve the timeliness of change-request processing during the maintenance phase of a system's life cycle. The resulting goal will specify a purpose (improve), a process (change-request processing), a viewpoint (project manager), and a quality

Table 2 Goal/Question/Metric Model

Goal	Purpose	Improve
	Issue	the timeliness of
	Object (process)	change-request processing
	Viewpoint	from the project manager's viewpoint
Question		Is the performance of the process improving?
Metrics		$\frac{\text{Current average turnaround time}}{\text{Baseline average turnaround time}}$
		Subjective rating of manager's satisfaction
Question		Is the distribution of resources changing?
Metrics		Percent effort spent on: • Problem analysis • Solution identification • Solution implementation • Solution testing

issue (timeliness). It then refines the goal into several questions that usually break the issue down into its major components. In the example we discuss later, the goal of the Software Engineering Laboratory can be refined to a series of questions about, for instance, turnaround time and resources used. It then refines each question into metrics. The questions in the example can be answered by metrics comparing specific turnaround times with an average. (The goal/question/metric model for our example is shown in Table 2.)

A company can also use the GQM approach for long-range corporate goal setting and evaluation. It can enhance the evaluation of a project by analyzing it in the context of several other projects. It can expand the level of feedback and understanding by defining the appropriate synthesis procedure for transforming specific, valuable information into more general packages of experience. In implementing the quality improvement paradigm, the company can formally learn more about the definition and application of the GQM approach, just as it would about any other experiences.

The Experience Factory: A Capability-Based Organization

In a capability-based organization, reuse of experience and collective learning become a corporate concern like the business portfolio or company assets. The experience factory is the organization that supports reuse of experience and collective learning by developing, updating, and providing, on request, clusters of competencies to be used by the project organizations.[7] We call these clusters of competencies "experience packages." The project organizations supply the experience factory with the products, plans, processes, and models used in their development and the data gathered during development and operation; the experience factory transforms them into reusable units and supplies them to the project organizations, together with specific monitoring and consulting support.

The experience factory's activities must be clearly identified and independent from those of the project organization. At the same time, the synergy and interaction between the experience factory and project organizations must be constant and effective. The project organization's goal is to produce and maintain software. The experience factory provides direct feedback to each project, together with goals and models tailored from similar projects. (Figure 3 shows the experience factory organization and highlights activities and information flows among the component suborganizations.)

The project organization provides the experience factory with project and environment characteristics, development data, resource usage information, quality records, and process information. This provides feedback on the actual performance of the models that the experience

Figure 3 Synergies between Project Organization and Experience Factory

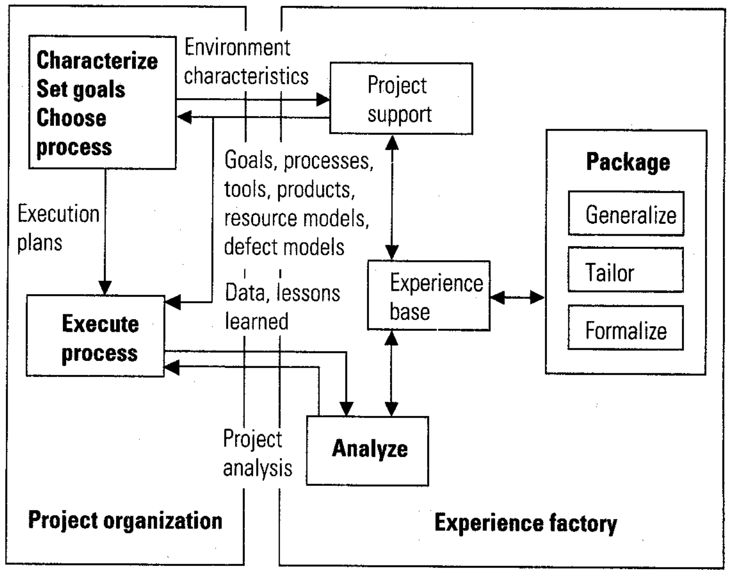


Table 3 Core Competencies and Corresponding Technologies

Core Competencies	Aggregate Technologies
• Use of an integrated software engineering environment tailored to one or more specific application domains	• Tool integration • Domain analysis and architectures • Data sharing and communication in heterogeneous environments
• Availability of reusable components (modules, algorithms, architectures) and tools portable across different platforms	• Reuse libraries, mechanisms, and methods • Domain analysis and architectures • Object-oriented techniques
• Availability and use of a software management environment based on "local" data for estimate, control, and prediction of projects	• Measurement and data collection and analysis • Data and process modeling • Defect counting, categorization, and analysis

factory processes and the project utilizes. The experience factory produces and provides baselines, tools, lessons learned, and data, parameterized in some form to adapt to a project's specific characteristics. Support personnel sustain and facilitate the interaction between developers and analysts by saving and maintaining the information, making it efficiently retrievable, and controlling and monitoring its access.

The main products of the experience factory are core competencies packaged as aggregates of technologies. (For some examples of core competencies and the corresponding aggregation of technologies, see Table 3.) A company can implement core competencies in various formats or experience packages. Their content and structure vary based on the kind of experience clustered within. There is generally a central element that determines what the package is, such as a software life-cycle product or process, an empirical or theoretical model, a database, and so on.

The synergy of the project organization and the experience factory is based on the quality improvement paradigm we introduced previously. Each component performs activities in all six steps, but, for each step, one component has a leadership role. (Figure 4 shows an outline of the whole organization and its mapping on the QIP.)

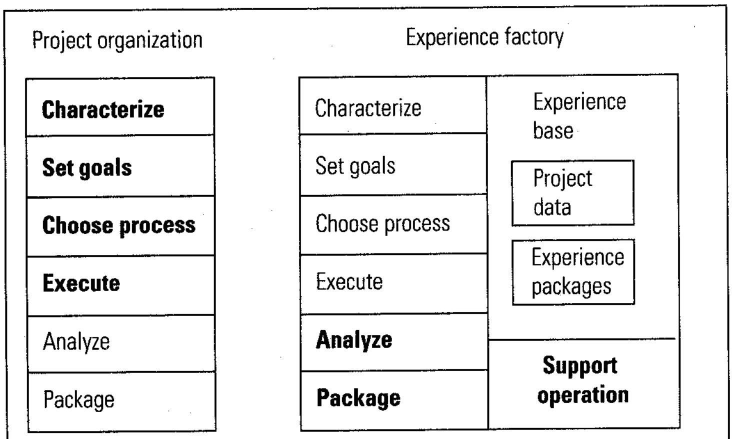
In the first three phases (characterize, set goals, and choose process), the operation focuses on planning. The project organization has a leading role and is supported by the experience factory analysts. The outcome of these three phases is, on the project organization side, a project plan associated with a management control framework, and on the experience factory side, a support plan also associated with a management control framework. The project plan describes the project's goals, phases, and activities, with their products, mutual dependencies, milestones, and resources. For the experience factory side, the plan describes the support that it will provide for each phase and activity and expected improvements.

In the fourth phase (execute), the operation focuses on delivering the product or service assigned to the project organization. The project organization again has a leading role, supported by the experience factory. The outcome of this phase is the product or service, which is associated with a set of potentially reusable products, processes, and experiences.

In the fifth and the sixth phases (analyze and package), the operation concentrates on capturing project experience and making it available to future similar projects. The experience factory has a leading role and is supported by the project organization that is the source of that experience. The outcomes of these phases are lessons learned with recommendations for future improvements, and new or updated experience packages incorporating the experience gained during the project execution.

Structuring a software development organization as an experience factory offers the ability to learn from every project, constantly increase the organization's maturity, and incorporate new technologies into the life cycle. In the long term, it supports the overall evolution of the organiza-

Figure 4 A Map of the Quality Improvement Paradigm for the Whole Organization



Note: Bold type indicates that the specific component has the lead on that specific task.

tion from project-based, where all activities are aimed at the successful execution of current project tasks, to capability-based, which capitalizes on task execution.

Table 4 Focus of the Software Engineering Lab's Three Components

Developers' Focus	Analysts' Focus	Support Infrastructure's Focus
Software development	Experience packaging	Support developers and analysts
Single application	Application domain	Organization
Decompose a problem into simpler ones	Generalize and formalize solutions and products	Categorize and organize
Tailor and apply the process	Analyze and synthesize the process	Store and retrieve the process information
Validation and verification	Experimentation	Efficient retrieval

An organization benefits from its structure as an experience factory by:

- Establishing a software improvement process substantiated and controlled by quantitative data.
- Producing a repository of software data and models that are empirically based on everyday practice.
- Developing an internal support organization that limits overhead and provides substantial cost and quality performance benefits.
- Providing a mechanism for identifying, assessing, and incorporating into the process new technologies that have proven valuable in similar contexts.
- Incorporating and supporting reuse in the software development process.

Improvement in Practice: A NASA Engineering Laboratory

Next we offer a practical example of an experience factory organization — the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center — and show how its operation uses the quality improvement paradigm.[8]

The SEL was established in 1976 as a cooperative effort among the Department of Computer Science of the University of Maryland, the National Aeronautic and Space Administration Goddard Space Flight Center (NASA/GSFC), and Computer Sciences Corporation (CSC). The lab's goal was to understand and improve key software development processes and products in a specific organization, the Flight Dynamics Division.

The goals, structure, and operation of the SEL have evolved from an initial stage — a laboratory dedicated to experimentation and measurement — to a full-scale organization aimed at reusing experience and developing strategic capabilities. The SEL's structure is based on three components:

- Developers, who provide products, plans used in development, and data gathered during development and operation (the project organization).
- Analysts, who transform the objects that the developers provide into reusable units and supply them to the developers; they support the projects on use of the analyzed, synthesized information, tailoring it for a current software effort (the experience factory proper).
- Support infrastructure, which provides services to the developers by supporting data collection and retrieval, and to the analysts by managing the library of stored information and its catalogs (the experience base support).

(For an outline of the differences in focus among the three suborganizations, see Table 4.)

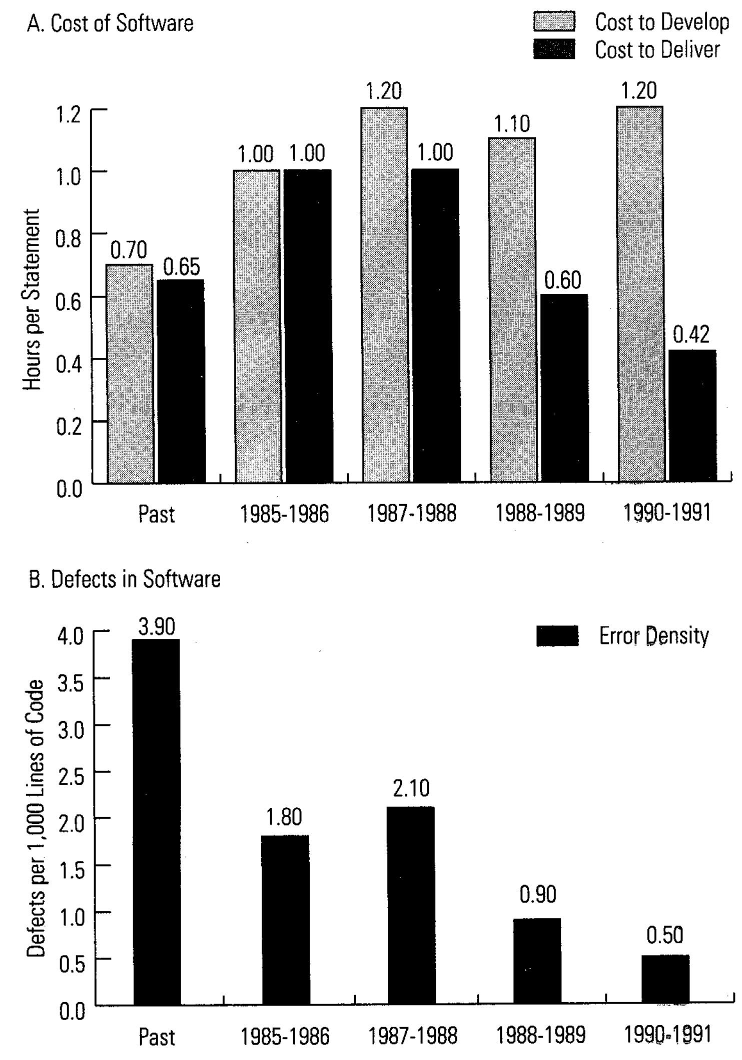
In the late 1980s, the software engineering community was considering the use of the Ada programming language environment and technology, which the U.S. Department of Defense had developed.[9] NASA thought of using Ada technology for some major projects such as the space station. Its application was also being considered in areas outside the Department of Defense. If more and more systems used Ada as a development environment, more organizations would be involved with it, and Ada would have to be transformed from simple technology to core competence for the software development organizations within NASA.

Associated with Ada was the issue of object-oriented technologies. Some basic characteristic elements of the object-oriented approach are:[10]

- A system is seen as a set of objects with a defined behavior and characteristics.
- Objects interact with each other by exchanging messages.
- Objects are organized into classes based on common characteristics and behaviors.
- All information about the state or the implementation of an object is held in the object itself and cannot be deliberately or accidentally used by other objects.

From the beginning, the SEL thought that the two technologies (Ada and object technology) should be packaged together into a core competence supporting the strategic capability of delivering systems with better quality and lower delivery cost. After it recognized that this capability had a strategic value for the organization, the SEL selected Ada and the object-oriented design technology for supporting it, measured its benefits, and provided data in support of its decision to use the technology.

Figure 5 Trends of Significant Indicators



The SEL followed these steps, according to the QIP:

1. Characterize. In 1985, the SEL developed a baseline of how the Flight Dynamics Division developed software. It defined the development processes and built models to improve the process's manageability. It integrated the standard development methodology, based on the traditional design-and-build approach, with concepts aimed at continuously evolving systems by successive enhancements.

2. Set goals. Realizing that object-oriented techniques implemented in the design and programming environments offered potential for major improvements in productivity, quality, and reusability of software products and processes, the SEL decided to develop a core competence around object-oriented design and Ada. First, it set up expectations and goals against which it measured results. The SEL's well-established baseline and measures provided an excellent basis for comparison. Its expectations included —

- An increase in effort on early phases of development activities (design) and a decrease on late phases (testing).
- Increased reuse of software modules.
- Decreased maintenance costs due to the better quality, reusable components.
- Increased reliability as a result of lower global error rates, fewer high-impact interface errors, and fewer design errors.

3. Choose process. The SEL decided to approach the development of the desired core competence by experimenting with Ada and object-oriented design in a "real" project. It developed two versions of the same system. System A used FORTRAN and followed the standard methodology based on functional decomposition. System B used Ada and followed an object-oriented methodology called HOOD. The SEL compared the data derived from the development of system B with those from system A. It devoted particular attention to quality and productivity data.

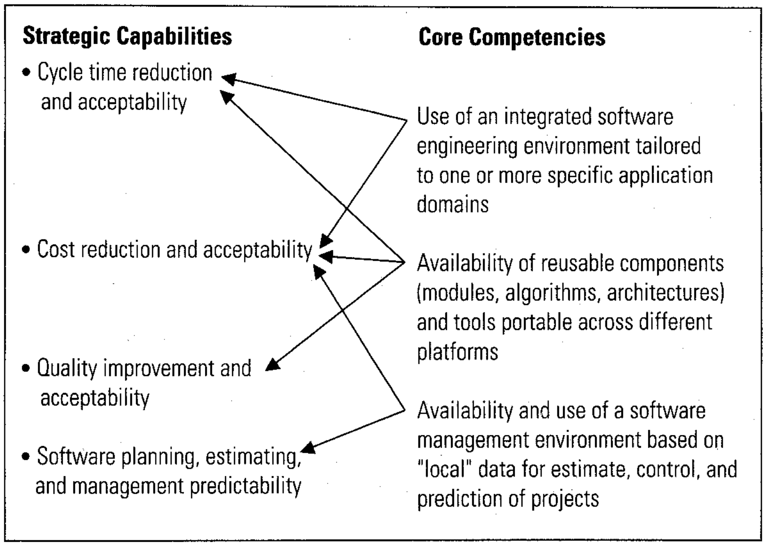
4. Execute. The SEL implemented systems A and B and collected the desired metrics.

5. Analyze. The data showed an increase in the cost to develop due to the organization's inexperience with the new technology and to the technology's intrinsic characteristics. The data also showed an increase in cost to deliver due to the same causes. The overall quality of system B showed an improvement over system A in terms of a substantially lower error density.

6. Package. The laboratory tailored and packaged an internal version of the methodology that adjusted and extended HOOD for use in a specific environment and on a specific application domain. Commercial training courses, supplemented with limited project-specific training, constituted the early training in the techniques. The laboratory also produced experience reports on the lessons learned using the new technology and recommendations for refinements to the methodology and standards.

Results of the Process. The data collected from the first execution of the process were encouraging, especially on

Figure 6 Relationships between Strategic Capabilities and Core Competencies


Strategic Capabilities
• Cycle time reduction and acceptability
• Cost reduction and acceptability
• Quality improvement and acceptability
• Software planning, estimating, and management predictability
Core Competencies
Use of an integrated software engineering environment tailored to one or more specific application domains
Availability of reusable components (modules, algorithms, architectures) and tools portable across different platforms
Availability and use of a software management environment based on "local" data for estimate, control, and prediction of projects

the quality issue, but inconclusive. The SEL decided on new executions to be continued in the future. Along with the development methodology, it developed a programming language style guide that provided coding standards for the local Ada environment.

The SEL has completed at least ten projects using an object-oriented technology derived from the one used for system B but constantly modified and improved. The size of single projects, measured in thousands of lines of source code, ranges from small to large. Some characteristics of an object-oriented development, using Ada, emerged early and have remained rather constant. No significant change has been observed, for instance, in the effort distribution or in the error classification. Other characteristics emerged later and took time to stabilize. Reuse has increased dramatically after the first projects, going from a traditionally constant figure of 30 percent reuse across different projects, to a current 96 percent (89 percent reuse). (See Figure 5.)

Over the years, use of the object-oriented approach and expertise with Ada have matured. Source code analysis of the systems developed with the new technology has revealed a maturing use of Ada's key features that has no equivalent in the programming environments NASA traditionally uses. The SEL used such features not only more often in more recent systems, but also in more sophisticated ways, as revealed by specific metrics for this purpose. Moreover, the use of object-oriented design and Ada features has stabilized during the past three years, creating an SEL baseline for object-oriented developments.

The cost to develop code in the new environment has remained higher than the cost to develop code in the old one. However, because of the high reuse rates obtained through the object-oriented paradigm, the cost to deliver a system in the new environment has significantly decreased and is now well below the old cost.

The reliability of the systems developed in the new environment has improved during the maturing of the technology. The error rates were significantly lower than the traditional ones and have continued to decrease. Again, the high level of reuse in the later systems is a major contributor to this greatly improved reliability.

Because of the technology's stabilization and apparent benefit, the object-oriented development methodology has been packaged and incorporated into the current technology baseline and is a core competence of the organization. Although the SEL will continue to refine the technology of object-oriented design, HOOD has now progressed through all stages, moving from a trial methodology to a fully integrated, packaged part of the standard methodology, ready for further incremental improvement.

The SEL example also illustrates the relationship between a competence (object-oriented technology) and a target capability (deliver high quality at low cost) and shows how innovative technologies can systematically enter the production cycle of mature organizations. Although the topic of technology transfer is not specifically within our scope here, it is clear that the model we derive from the SEL example outlines a solution to some major technology-transfer issues. The purpose of an experience factory organization, however, goes beyond technology transfer to encompass capability transfer and reuse.

Conclusion

For software, the remainder of the 1990s will be the era of quality and cycle time. There is a growing need to develop or adapt quality improvement approaches to the software business. Our approach to software quality improvement is based on the exploitation and reuse of an organization's critical capabilities across different projects based on business needs.

The relationship between core competencies and strategic capabilities is established by the kind of products and services the organization wants to deliver and is specified by the strategic planning process. (Figure 6 gives a possible map for an organization whose main business is systems and software development for user applications.) The SEL example shows that these ideas are feasible and have been successfully applied in a production environment to create a continuously improving organization. Such an organization can manipulate its processes to achieve various product characteristics. It needs to have a process and organizational structure to:

- Understand its processes and products.
- Measure and model its business processes.
- Define process and product quality explicitly and tailor the definitions to the environment.
- Understand the relationship between process and product quality.
- Control project performance with respect to quality.
- Evaluate project success and failure with respect to quality.
- Learn from experience by repeating successes and avoiding failures.

By using the quality improvement paradigm/experience factory approach, an organization has a good chance to achieve all these capabilities and improve quality faster because it focuses on its strategic capabilities and value-added activities. The experience factory organization is the lean enterprise model for the system and software business. ◆

References

We acknowledge the contributions of all those who participated in the experiences and discussions that originated the concepts presented here. Particular acknowledgment goes to the personnel of the Software Engineering Laboratory at NASA Goddard Space Flight Center and Frank McGarry, Jerry Page (CSC), Tony Jordano (SAIC), Bob Yacobellis (Motorola), Paolo Sigillo (Italsiel), and Mike Deutsch (Hughes Information Technology Corporation).

1. W. Edwards Deming, *Out of the Crisis* (Cambridge, Massachusetts: MIT Press, Center for Advanced Engineering Study, 1986).

2. A.V. Feigenbaum, *Total Quality Control* (New York: McGraw Hill, 1991).

3. J.P. Womack, D.T. Jones, and D. Roos, *The Machine That Changed the World* (New York: Rawson Associates, 1989).

4. G. Stalk, P. Evans, and L.E. Shulman, "Competing on Capabilities: The New Rules of Corporate Strategy," *Harvard Business Review*, March-April 1992, pp. 57-69.

5. V.R. Basili, "Quantitative Evaluation of a Software Engineering Methodology" (Melbourne, Australia: Proceedings of the First Pan-Pacific Computer Conference, September 1985); and
V.R. Basili, "Software Development: A Paradigm for the Future" (Orlando, Florida: Proceedings of COMPSAC '89, September 1989), pp. 471-485.

6. V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984, pp. 728-738; and
V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988, pp. 758-773.

7. Basili (1989).

8. V.R. Basili, G. Caldiera, F. McGarry, R. Pajerski, J. Page, and S. Waligora, "The Software Engineering Laboratory — An Operational Software Experience Factory" (Melbourne, Australia: Proceedings of the Fourteenth International Conference on Software Engineering, May 1992).

9. ANSI/MIL-STD-1815A 1983: *Reference Manual for the Ada Programming Language*.

10. I. Sommerville, *Software Engineering* (Wokingham, England: Addison-Wesley, 1992).

Reprint 3715