THE EXPERIENCE FACTORY
STRATEGY AND PRACTICE

Victor R. Basili
basili@cs.umd.edu

Gianluigi Caldiera
gcaldiera@cs.umd.edu

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

The quality movement, that has had in recent years a dramatic impact on all industrial sectors, has recently reached the systems and software industry. Although some concepts of quality management, originally developed for other product types, can be applied to software, its specificity as a product which is developed and not produced requires a special approach. This paper introduces a quality paradigm specifically tailored on the problems of the systems and software industry.

Reuse of products, processes and experience originating from the system life cycle is seen today as a feasible solution to the problem of developing higher quality systems at a lower cost. In fact, quality improvement is very often achieved by defining and developing an appropriate set of strategic capabilities and core competencies to support them. A strategic capability is, in this context, a corporate goal defined by the business position of the organization and implemented by key business processes. Strategic capabilities are supported by core competencies, which are aggregate technologies tailored to the specific needs of the organization in performing the needed business processes. Core competencies are non-transitional, have a consistent evolution, and are typically fueled by multiple technologies. Their selection and development requires commitment, investment and leadership.

The paradigm introduced in this paper for developing core competencies is the Quality Improvement Paradigm which consists of six steps:

1. *Characterize the environment*
2. *Set the goals*
3. *Choose the process*
4. *Execute the process*
5. *Analyze the process data*
6. *Package experience*

The process must be supported by a goal-oriented approach to measurement and control, and an organizational infrastructure, called Experience Factory. The Experience Factory is a logical and physical organization distinct from the project organizations it supports. Its goal is development and support of core competencies through capitalization and reuse of life cycle experience and products.

The paper introduces the major concepts of the proposed approach, discusses their relationship with other approaches used in the industry, and presents a case in which those concepts have been successfully applied.

This work was supported by NASA Grant NSG-5123 and by Hughes Applied Information Systems, Inc.

1. INTRODUCTION

The presence of software in almost every activity and institution is a characteristic of our society. Our dependence on software becomes evident when software problems and related events make the headlines of newspapers. However, this dependency on software, although highly visible, is not yet well understood by the business community. Software is still too often perceived as the easiest part of a system, the part that can be easily modified and adapted to fit to the main business of the organization.

This idea that "software is easy" or, ultimately, "cheap" is hard to eradicate, even when there is substantial evidence that it is not true anymore. In particular, there is a certain difficulty in dealing with software quality, both it terms of definition (What is quality software?) and implementation of quality programs (How can we produce quality software?).

The starting point of every discussion on software quality is the recognition that software is an industrial product whose quality can be managed in a similar way to the quality of other products or services. A software system is the result of the concurrent effort of teams of people working according to a traditional engineering paradigm (a conception phase followed by an implementation phase, very often with several iterations). In fact, we call "software engineering" the systematic approach to the development, operation and maintenance of software systems (and associated documentation and data).

As with every industrial product, the quality of software is defined as "fitness for use" over its lifetime. Therefore, the goal of a quality management program is to incorporate quality into a software system in the most economically convenient way, i.e., by designing a high quality system. The challenge of software quality is to implement techniques and programs in order to fill the existing gap between demand and our ability to produce high-quality software in a cost-effective way.

The software product, however, presents the following critical combination of characteristics:

- *Software is a logical aggregate of invisible parts*: The quality of such aggregate depends on the appropriateness of the logical structuring of the parts and on a precise and easy-to-understand documentation of this structure;

- *Software is designed for user applications which are expected to evolve continuously*: The quality of application software depends on the precise conceptual understanding of user needs, and on the adaptability of design to a changing environment; good communication between designers and users, and user perception are essential components of good software design;

- *Software is developed and not produced*: Each software product is like a prototype, therefore many statistical concepts that help us in measuring and controlling quality in industrial products do not apply completely to software products;

- *Software is a human based technology*: The quality of the software product is dependent on the individuals involved, therefore appropriate use of individual skills, individual satisfaction and motivation are key issues in achieving substantial improvements in quality and productivity.

We believe that the quality of a software system should and can be managed in two ways. First, the effectiveness of the software development process should be improved by reducing the amount of rework and reusing software artifacts across segments of a project or different projects. Second, plans for controlled, sustained, and continuous improvement should be developed and implemented based on facts and data.

But software engineering does not make extensive use of quantitative data. Therefore software quality management is based on a very immature and unstable paradigm. A major problem is that many data regarding the quality of a system can only be observed, and measured when the system is implemented. Unfortunately, at that stage the correction of a design defect requires the redesign of some, sometimes large and complex, components and is very expensive. In order to prevent the occurrence of expensive defects in the final product, quality management must focus on the early stages of the engineering process, in particular on the requirements analysis and design phases, and use quantitative data in order to record and support inspection and decision making. Those early stages are, however, the ones in which the process is less defined and controllable with quantitative data. Therefore, software engineering projects do not regularly collect data and build models based upon them.

There are many software project that can be considered successful from a quality point of view; generally this means that the techniques and procedures applied in the project have been effective, in particular those aimed at assuring quality. The goal of quality management is to make this success repeatable in other projects, by transferring the knowledge and the experience that are at the roots of that success to the rest of the organization. Therefore, a software organization that manages quality should have, besides the quality assurance infrastructure associated with each project, a corporate infrastructure that links together and transcends the single projects by capitalizing on successes and learning from failures.

Quality management and infrastructure, however, do not just happen; they must be planned and implemented by the organization through specific programs and investments. This paper is about the need for a strategic approach to software quality management, as a part of a corporate strategy for software, aimed at pursuing and improving quality as an organization and not as a group of individual projects.

We will motivate the need for such an approach, discuss it in the context of some of the most relevant concepts developed by the management disciplines, and provide a framework for a solution, which has been applied in practice with convincing results.

We believe there is no solution that can be mechanically transferred and applied to every organization (the famous "silver bullet"), and this applies also to the concepts presented in this paper. The proposed approach, however, can be used by every organization, after appropriate customization, in order to improve software quality in a controllable way.

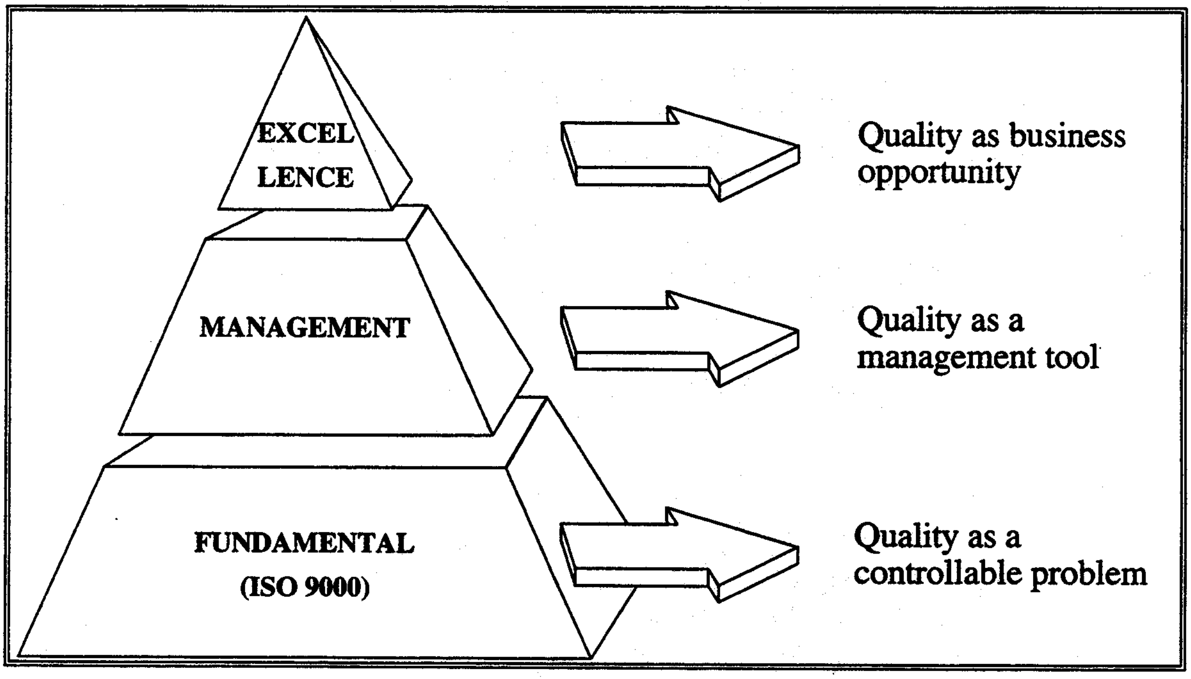
2. THE PROBLEM OF SOFTWARE QUALITY

Quality is the totality of characteristics of a product or service "that bear on its ability to satisfy stated or implied needs" [ISO1]. It is a multidimensional concept that includes the entity of interest (the product or service), the viewpoint on that entity (the user, the producer, a regulatory agency, etc.) and the quality attributes of that entity (the characteristics that make it fit for use). A recent international standards [ISO3] identifies the following characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

In some cases, such as regulated environments in which some safety critical factors must be determined (aeronautics, nuclear power, etc.), these attributes are specified by a standard or a contract; but in the majority of cases they are identified and defined during the design process, and modified throughout the life cycle of the system. The ability of an organization to identify and define the quality attributes that are closer to the "stated or implied needs" of a user is the critical success factor in the market of the 90's.

Figure 1


EXCEL LENCE
MANAGEMENT
FUNDAMENTAL (ISO 9000)
Quality as business opportunity
Quality as a management tool
Quality as a controllable problem

Today the success of a software organization is measured by its cost/performance attributes: it delivers (or updates) the needed systems generally on time and without budget overruns. In the longer run, though, if we take into account today's market, characterized by shrinking budgets and increased global competition, we can expect, for the second half of the '90s, that the most successful organizations will probably be the ones that have been able to converge to better levels of productivity and quality. The influence of international standards such as the ISO 9000 Series [ISO2] is already evident. Many organizations are now seeking registration and the ability to develop quality systems in compliance with the requirements of the standard. Registration, however, is a means and not an end: spending resources on developing a quality system without a quality improvement program that uses it to gain a competitive advantage would be a waste of money. This is why, along with ISO 9000 registration programs, we see quality improvement programs being started. We can expect that in a few years all this movement will lead to a higher quality baseline for all the software that is being purchased and developed around the world. On top of this baseline the organizations will be able to build their own quality management programs and their continuous improvement strategies. In this way quality will complete its transformation from problem (search for defects) to tool (defined processes) to business opportunity used to distinguish an organization from its competitors(Figure 1).

At that point, the real advantage will come from the ability of the software organization to deliver solutions that not only satisfy, but also anticipate the needs of the system users, enhancing their business and adding a substantial amount of value to their products and services [Hamel and Prahalad, 1991]. Competition in the '90s is a more complex and dynamic playing field, in which the basic factors for success are the understanding of trends and the response to changing needs. The traditional rigidity of software organizations must to be adapted to the new ground rules. New professional skills, beyond the traditional programmer/analyst/manager triangle, are necessary in order to capitalize on the experience of the organization and work on specific lines of business instead of developing isolated products.

If we survey the approaches to software quality available to the industry, we see a variety of paradigms, mostly coming from the manufacturing industry.

Some organizations apply to their software processes an improvement process based on the Shewart-Deming Cycle [Deming, 1986]. This approach provides a methodology for managing change throughout the steps of a production process by analyzing the impact of those changes on the data derived from the process. The methodology is articulated in four phases:

- Plan: Define quality improvement goals and targets and determine methods for reaching those goals; prepare an implementation plan.
- Do: Execute the implementation plan and collect data.
- Check: Verify the improved performance using the data collected from the process and take corrective actions when needed.
- Act: Standardize the improvements and install them into the process.

Some organizations use the Total Quality Management (TQM) approach, which is a derivative of the PDCA method applied to all business processes in the organization [Feigenbaum, 1991]. Actually, more than a specific method TQM is a family of management philosophies based on the fact that quality is measured by the user of a product, and that everyone in the organization has specific responsibilities for the quality of the final outcome. Therefore, in TQM programs, quality improvements, identified during a preliminary characterization effort, are usually experimented by pilot groups and then institutionalized across the whole organization. The TQM approach usually results in the establishment of cross-functional quality improvement teams chartered to addressing specific quality improvements within a strategic quality plan developed by the top management.

A different approach is adopted by organizations that model their improvement on an external scale that is meant to represent the best practices in quality. The goals of the improvement program are, in this case, not internally generated but suggested by those best practices. A model of this kind, which is today very popular in both the USA and Europe, is the SEI Capability Maturity Model [SEI; Bootstrap] which measures the maturity of a software organization on the basis of its dependence on individual skills and on the presence of certain technologies. In a low maturity organization, the success of a task depends on the efforts of people involved in it, professionals and managers. Their ability to control risk, to solve or even prevent problems is the major asset of the organization. In a more mature organization, the success is based on the use of sound managerial and engineering techniques coordinated by a pervasive, well-defined set of processes for the execution of the needed tasks. At the highest level of maturity, the organization effectively capitalizes on its experiences and improves its processes. The improvement is achieved by bringing the organization through these levels of maturity.

All these approaches, and variations on them, have been used by the software industry, with mixed outcomes. Some outstanding successes have been reported, such as the one shown in Figure 2 [Dion, 1993], by combining those approaches. The major problem with all these approaches is that they either do not deal specifically with the nature of the software product (Deming Cycle, TQM) or, if they do, they assume that there is a consistent picture of what a good software product or process is (SEI model).

We argue that this is not enough for two reasons: the first one is that in order to be really effective a software quality program should deal with the nature of the software business itself; the second is that there is really no such thing as an explicit consistent picture of a good software product.

Figure 2

Raytheon Experience

Costs

- $ 1 million: Investment on the improvement program for each year (1987-1992)

Benefits

- $ 15.8 million: Rework costs eliminated

Return on investment

- 7.7 : 1 in 1990

Changes in % project time by cost type from 1988 to 1990

- Performance: Cost of building it right the first time, from 34% to 55%;
- Non conformance: Cost of rework, from 44% to 18%;
- Appraisal: Cost of testing, from 15% to 15%;
- Prevention: Cost of preventing non-conformance, from 7% to 12%.

On one hand, if we look at processes and technologies in isolation, like in the Plan/Do/Check/Act and TQM approaches, we have very little chance to get to the right level of abstraction that provides reusable units across different processes. Those approaches do not really build "model abstractions" because they manipulate the process explicitly. For instance: if we apply TQM to the order entry process, we have well defined elementary actions performed to enter an order. We can describe them with a flow chart and analyze the process, apply changes and assess their impact. We will have very soon many instances of that process to build a control chart and bring it under control. Unfortunately, the same approach cannot be used on a software process (e.g., structured design), which cannot be reduced to elementary units and is not replicated many times in a short period.

On the other hand, if we base our judgment upon an external model, like in the SEI and similar approaches, we might loose characteristics that make an organization's environment "special." Those characteristics are, in many cases, at the roots of the competitive advantage of that organization, therefore their loss is very damaging for the improvement program.

The approach that will be presented in the next sections of this paper is an attempt to learn from the successes obtained through the different paradigms sketched in this section, and to avoid the problems encountered in their application to software environments. It rests on the *lean enterprise concept* [Womack, 1989] by concentrating production and resources on value-added activities that represent the critical business processes of the organization. Such processes, after having been recognized, are conceptually redesigned in a modular way and associated with models, data, techniques and tools, in order to reuse them according to the needs and characteristics of specific projects. Total quality management [Feigenbaum, 1991] and Concurrent engineering [Dewan and Riedl, 1993] can be used in order to keep the structure efficient, responsive to the needs of any external entity (customer or supplier), and to make it rest upon partnership and participation, with many feedbacks and measures of the effectiveness of communication.

3. TOWARDS A MATURE SOFTWARE ORGANIZATION

If we analyze carefully some of the most successful and trend-setting business stories of the last 10 years [Stalk, Evans and Shulman, 1992], we can ascribe the reported successes to the application of four basic principles:

1. Business processes are the building blocks of the corporate strategy.

2. Competitive success depends on understanding and transforming the key business processes into strategic capabilities.

3. Strategic capabilities are created by sustained long-term investments in a support infrastructure that links together and transcends the business units.

4. A capability-based strategy must be sponsored by the top management of the corporation.

It is important to understand these four principles in the context of on a software organization.

The first principle sets the focus on business processes: this is consistent with the current tendency to emphasize the role of software processes in a successful project. Software is a logical aggregation and an intellectual product, which is, therefore, strongly dependent on the processes executed for developing or maintaining it. The analysis of those processes and the ability to reuse them in the appropriate context are a key competitive factor for every software organization. The corporate strategy must focus on identification and characterization of the key business processes used in developing and maintaining software, so that the business units, relieved from process related concerns, can focus more on the individual systems and services that are developed and delivered to individual clients.

The second principle is about "strategic understanding" of business processes. This means that the organization must understand its key business processes sufficiently to transform them into reusable units available to all its business units where needed. Not every process used in the organization has the characteristics of criticality that make it worthy of being transformed into a strategic capability: it is only from the analysis of the relationship between software processes and the mission of the organization that we can obtain a strategic level of understanding and a consolidated hypothesis of what should become a strategic capability. A system developer or integrator, for instance, produces software in order to deliver services to a particular group of users (e.g., electronic messaging). In this case a good cost/benefit ratio for the system or service is probably the most crucial issue. Therefore, the process of making acceptable estimates and to develop a plan based on them has a criticality definitely higher than the process of assuring the highest possible reliability. On the other hand, for a manufacturer of systems dependent on software (e.g., cellular phones) the cost/benefit ratio for software is distributed over a large number of products and therefore not extremely crucial for the single software package. Therefore, the process of assuring reliability has a higher criticality in comparison with the ability of making acceptable estimates of software costs.

The third and the fourth principles call for long-term investments and top management sponsorship, which translates into a permanent structure that develops and supports the reuse of the strategic capabilities. This is particularly new for the software industry, which is, in its large majority, driven by its business units and, therefore, has little ability to capitalize on experiences and capabilities. The required permanent structure is designed to provide a double support cycle:

- Control cycle: Support is provided to the everyday operation of software projects by comparing their current performance with the normal performance of similar projects;

- Capitalization cycle: Support is provided to future projects by continually learning from past experience and packaging this experience in a reusable way.

The development of strategic capabilities and competencies to support them, which is the key to all four of the presented principles, has, in the case of software, some basic requirements:

1. The organization must understand the software process and product.

2. The organization must define its business needs and its concept of process and product quality.

3. The organization must evaluate every aspect of the business process, including previous successes and failures.

4. The organization must collect and use information for project control.

5. Each project should provide information that allows the organization to have a formal quality improvement program in place, i.e. the organization should be able to control its processes, to 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 organization's business.

Part of the problem with the software business is the lack of understanding of the nature of software and software development. To some extent, software is different from most products. First of all, software is developed in the creative, intellectual sense, rather than produced in the manufacturing sense, i.e., each software system is developed rather than manufactured. Second, there is a non-visible nature to software. Unlike an automobile or a television set, it is hard to see the structure or the function of software, or to reason about it in a straightforward way. Therefore, the development of strategic capabilities in software requires understanding, model building and continuous feedback from the process.

This means that we must rethink the software business and expand our focus to a new set of problems and the techniques needed to solve them. Unfortunately, the traditional orientation of a software project is based on a case-by-case problem solving attitude; the development of strategic capabilities is based, instead, on an experience reuse and organizational sharing attitude. Figure 3 outlines the traditional focus of software development and problem solving, along with the expanded focus, proposed here for experience reuse.

The obvious question to be asked now is: are there any practical models that can be used in order to develop a strategy with the new focus? Such practical models can be software organizations that have tried to implement a capability-based strategy (or at least parts of it) and have carefully collected lessons learned and data, empirical studies in-the-large based on the scientific method (observe, formulate a hypothesis, measure and analyze, validate/refute the hypothesis) that have published their findings in a workable form, controlled experiments in-the-small.

Figure 3

Traditional Focus	New Extended Focus
• Delivering specific products and services	• Developing capabilities
• Decomposing a complex problem into simpler ones	• Unifying different solutions into more general ones
• Design/implementation process	• Analysis/Synthesis process
• Instantiation	• Generalization and formalization
• Validation and verification	• Experimentation

In Section 5 we will illustrate an experience that we, together with large part of the software engineering community, consider a practical model. The reason for choosing this one, besides the personal involvement of the authors of this paper with it, which provides us with considerable insight, is its almost unique blend of an organizational strategy aimed at continuous improvement, of a data-based approach to decision making, of an experimental paradigm, along with many years of continuous operation and data collection.

4 A STRATEGY FOR IMPROVEMENT

This section will present a strategy for improvement based on the development of strategic capabilities.

The main concept of this strategy is the central role played by a methodological framework addressing the development and improvement of strategic capabilities in form of reusable experience. This framework will be presented and discussed in the form of a process called "Quality Improvement Paradigm" [Basili, 1985]. In order to manage this conceptual framework we will need two tools

- A control tool: The goal-oriented approach to measurement addressing the issue of supporting the improvement process with quantitative information [Basili and Weiss, 1984];
- An organizational tool: An infrastructure aimed at capitalization and reuse of software experience and products [Basili, 1989].

In the next section we will see the methodological framework and the associated tools at work in a specific and practical example.

4.1 THE QUALITY IMPROVEMENT PARADIGM

A *strategic capability* is for us a corporate goal defined by the business position of the organization and implemented by key business processes. Strategic capabilities of software organizations are identified by the analysis of the categories of products/services that the organization intends to deliver in the future, of the level of project control needed in order to deliver those products/services at the appropriate level of quality, and of the strengths and weaknesses of the organization. Examples of strategic capabilities are

- Certify the reliability of the system that is being released for acceptance by the customer;
- Have a design-to-cost process, i.e., tailor the design of a software system to the amount of available resources (money, people, computers, etc.);
- Use flexible standards, i.e. standards that can, case by case, be tailored to the needs and the characteristics of each project;
- Have a short cycle-time, i.e., reduce the elapsed time from the identification of a solution to its deployment.

Strategic capabilities are always supported by *core competencies,* which are aggregate technologies tailored to the specific needs of the organization in performing the needed business processes. For instance: in order to certify the reliability of a system, an organization needs to master the quality assurance process owning competencies such as statistical testing and reliability modeling; in order to design to cost the organization must use flexible processes owning competencies such as process modeling and control, and concurrent engineering.

Core competencies have characteristics that distinguish them from simple technologies or clusters of technologies:

- They are non-transitional: although sometimes they appear to be fashionable concepts, they don't come and go;
- They have a consistent evolution: a paradigm for their interpretation and application is built over time and some consensus is generated throughout the user community;
- They require commitment, investment and leadership;
- They are typically fueled by and work with multiple technologies;
- They generally support multiple product/service lines.

The acquisition of core competencies that support the strategic capabilities is the goal of the process we will present in this section. If a competency is a key factor in a strategic capability, the organization must be sure to own, control and properly maintain this competency at state-of-the-art level, and know how to tailor it to the characteristics of specific projects and business units.

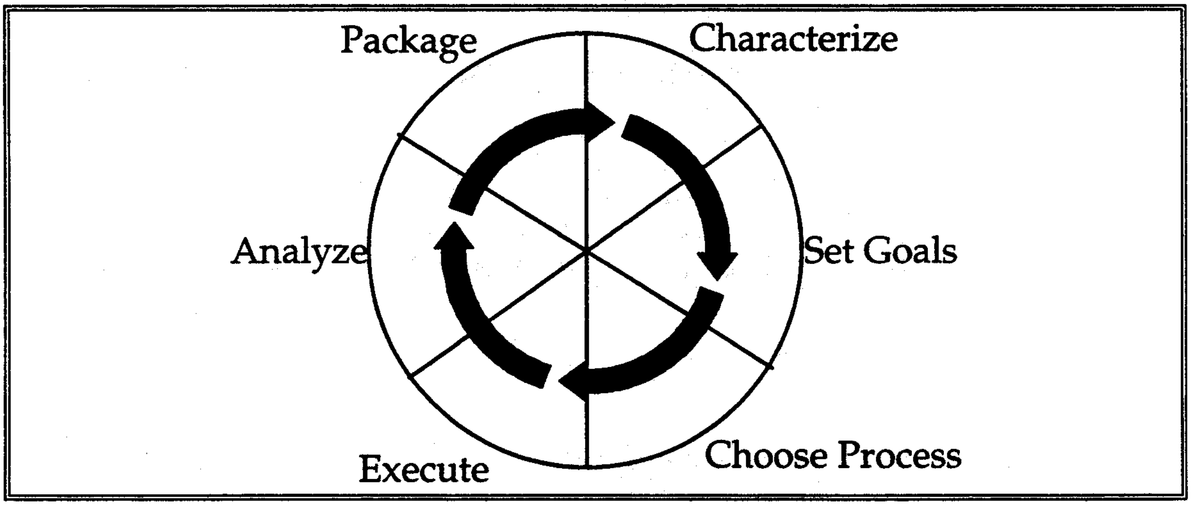
Strategic capabilities come into the improvement process as constituents of characteristics and goals. On the basis of the characteristics of the environment and of the transformation of those capabilities into specific goals for the software organization, the improvement paradigm provides a disciplined way to build the competencies necessary to support those capabilities.

The improvement process is articulated into the following six steps (Figure 4):

1. *Characterize*: Understand the environment based upon available models, data, intuition, etc. Establish baselines with the existing business processes in the organization and characterize their criticality.

2. *Set Goals*: On the basis of the initial characterization and of the capabilities that have a strategic relevance to the organization, set quantifiable goals for successful project and organization performance and improvement. The reasonable expectations are defined based upon the baseline provided by the characterization step.

Figure 4



3. *Choose Process*: On the basis of the characterization of the environment and of the goals that have been set, choose the appropriate processes for improvement, and supporting methods and tools, making sure that they are consistent with the goals that have been set.

4. *Execute*: Perform the processes constructing the products and providing project feedback based upon the data on goal achievement that are being collected. The processes will be

executed according to the needs dictated by the problem and to the process chosen in the previous phase.

5. *Analyze*: At the end of the execution, analyze the data and the information gathered to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.

6. *Package*: Consolidate the experience gained in the form of new, or updated and refined, models and other forms of structured knowledge gained from this and prior projects, and store it in an experience base so it is available for future projects.

The Quality Improvement Paradigm implements the two major cycles, control and capitalization, introduced in section 3:

- The project feedback cycle (control cycle) is the feedback that is provided to the project during the execution phase: whatever the goals of the organization, the project should use its resources in the best possible way; therefore quantitative indicators at project and task level are useful in order to prevent and solve problems, monitor and support the project, realign the process with the goals;

- The corporate feedback cycle (capitalization cycle) is the feedback that is provided to the organization and has the purpose of

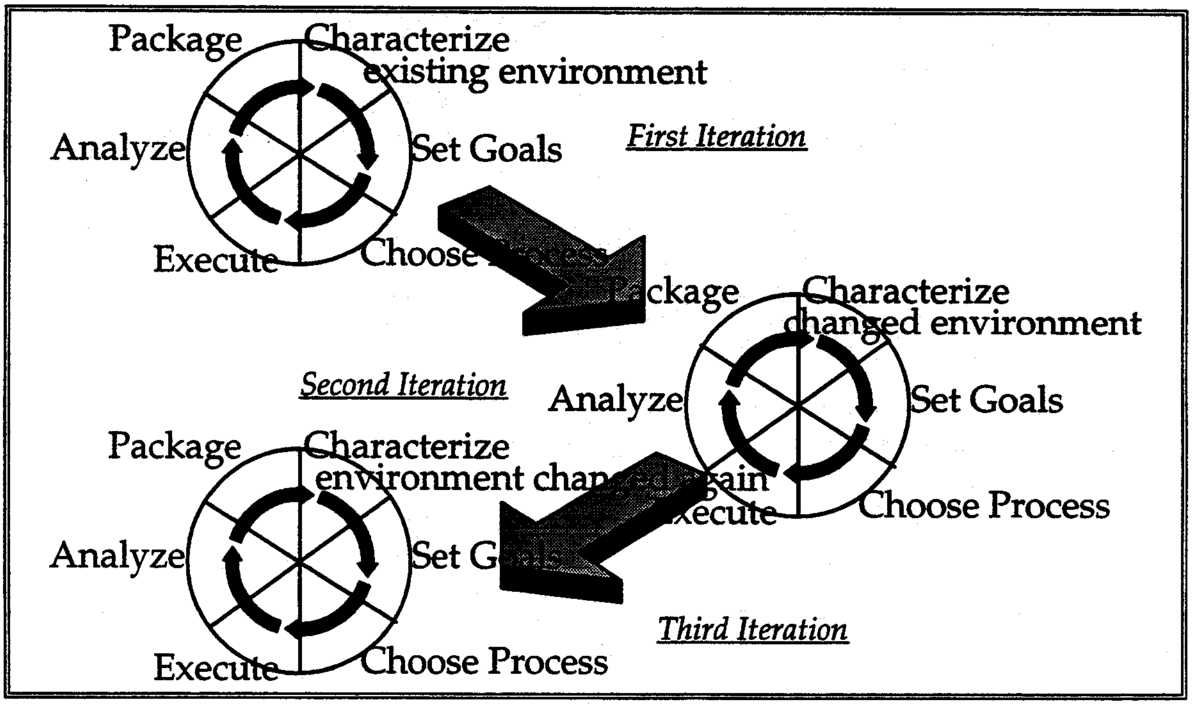
    - Providing analytical information about project performance at project completion time by comparing the project data with the nominal range in the organization and analyzing concordance and discrepancy;

    - Understanding what happened, capturing experience and devising ways to transfer that experience across domains;

    - Accumulating reusable experience in the form of software artifacts that are applicable to other projects and are, in general, improved based on the performed analysis.

The execution of the quality improvement paradigm by an organization is structured as an iterative process that repeatedly characterizes the environment, sets appropriate goals and chooses the process in order to achieve those goals, then proceeds with the execution and the analytical phases. At each iteration characteristics and goals are redefined and improved (Figure 5).

Figure 5



The reader has probably realized at this point that there is a deep similarity between the QIP and the Total Quality Management (TQM) philosophy. Figure 6 outlines some other correspondences between the two models.

The relationship between the QIP and the Plan/Do/Check/Act cycle is even closer. Both approaches are an offspring of the modern scientific method: first an hypothesis is generated, then an experiment is planned in order to validate the hypothesis, data are collected and analyzed, and the hypothesis is evaluated. The concept of feedback is also critical to both approaches: during the execution of the processes that have been planned and at the end of the execution data are analyzed in order to understand the impact of the changes introduced into the process. The real major difference between the two approaches appears at the end of the cycle: the PDCA approach incorporates the changes into the normal operation of the process, while the QIP develops a series of models that reflect the changes. This is due, as we said before, to the relatively smaller number of process instances that we have in the case of a software process, when compared with a manufacturing process.

Figure 6

TQM	QIP
Total Quality Management	**Quality Improvement Paradigm**
• Implements a corporation-wide quality improvement program	• Implements a program for reuse and improvement of software experience, artifacts, and processes
• Focuses on customer satisfaction and partnership for quality	• Focuses on customer satisfaction and partnership for quality
• Customers are both external and internal to the organization	• Capitalizes on project achievements
	• Customers are both external and internal to the organization
• Develops a flexible corporate culture	• Incorporates flexibility into the software process and product
• Bases decision making on facts	• Bases decision making on facts and data collected across different projects

4.2 THE GOAL-ORIENTED MEASUREMENT

The Goal/Question/Metric Approach [Basili and Weiss, 1984; Basili and Rombach, 1988] provides a method to identify and control key business processes in a measurable way. It is used to define metrics over the software project, process and product in such a way that the resulting metrics are tailored to the organization and to its goals, and reflect the quality values of the different viewpoints (developers, users, operators, etc.).

The result of the application of the Goal/Question/Metric Approach is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model has three levels:

1. Conceptual level (GOAL): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from

various points of view, relative to a particular environment. Objects of measurement include

- Products: Artifacts, deliverables and documents that are produced during the system life cycle; E.g., specifications, designs, programs, test suites.

- Processes: Software related activities normally associated with time; E.g., specifying, designing, testing, interviewing.

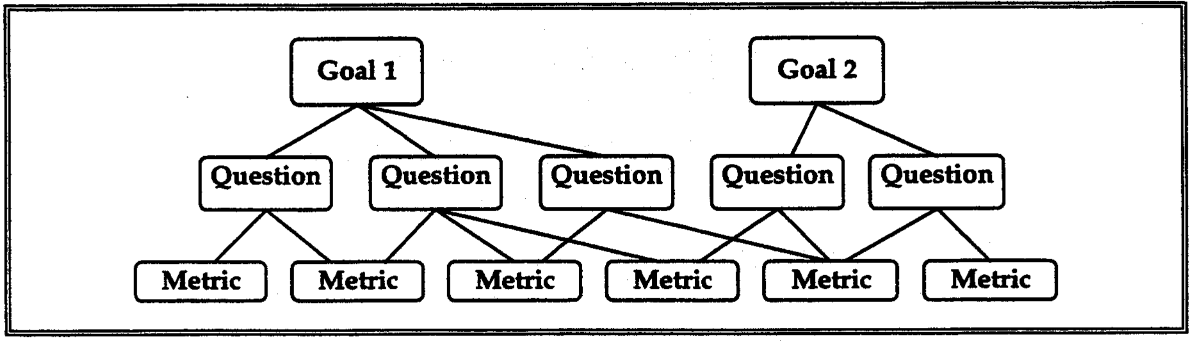
- Resources: Items used by processes in order to produce their outputs; E.g., personnel, hardware, software, office space.

- Knowledge objects: Models of the behavior of other items derived from past observations; E.g., resource models, reliability models.

2. Operational level (QUESTION): A set of questions is used to define in a quantitative way the goal and to characterize the way the specific goal is going to be interpreted based on some characterizing model. Questions try to characterize the object of measurement (product, process, resource, knowledge object) with respect to a selected quality issue and to determine its quality from the selected viewpoint.

3. Quantitative level (METRIC): A set of data is associated with every question in order to answer it in a quantitative way.

Figure 7



A GQM model is a hierarchical structure (Figure 7) starting with a goal (specifying purpose of measurement, object to be measured, issue to be measured, and viewpoint from which the measure is taken). In order to give an example of application of the Goal/Question/Metric approach, let's suppose we want to improve the timeliness of change request processing during the maintenance phase of the life-cycle of a system. The resulting goal will specify a purpose (improve), a process (change request processing), a viewpoint (project manager), and a quality issue (timeliness) (Figure 8). The goal is refined into several questions that usually break down the issue into its major components. The goal of the example can be refined to a series of questions, about, for instance, turn-around time and resources used. Each question is then refined into metrics. The questions of our example can, for instance, be answered by metrics comparing specific turn-around times with the average ones. The same metric can be used to answer different questions under the same goal. Several GQM models can also have questions and metrics in common, making sure that, when the measure is actually taken, the different viewpoints are taken into account correctly (i.e., the metric might have different values when taken from different viewpoints). The Goal/Question/Metric Model of our example is shown in Figure 8.

Figure 8

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 Percent effort spent on solution identification Percent effort spent on solution implementation Percent effort spent on solution testing

In conclusion, we can also use the Goal/Question/Metric Approach for long range corporate goal setting and evaluation. The evaluation of a project can be enhanced by analyzing it in the context of several other projects. We can expand our level of feedback and understanding by defining the appropriate synthesis procedure for transforming specific, valuable information into more general packages of experience. As a part of the Quality Improvement Paradigm, we can learn more about the definition and application of the Goal/Question/Metric Approach in a formal way, just as we would learn about any other experiences.

4.3 EXPERIENCE FACTORY: THE CAPABILITY-BASED ORGANIZATION

The concept of the Experience Factory [Basili, 1989] has been introduced in order to institutionalize the collective learning of the organization that is at the root of continuous improvement and competitive advantage.
Reuse of experience and collective learning cannot be left to the imagination of single, very talented, managers: in a capability-based organization they become a corporate concern like the portfolio of businesses or the company assets. *The experience factory is the organization that supports reuse of experience and collective learning by developing, updating and providing upon request clusters of competencies to the project organizations* . We call these clusters of competencies, experience packages. The project organizations supply the experience factory with their products, the 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 support made of monitoring and consulting.

The experience factory organization can be a logical and/or physical organization, but it is important that its activities are clearly identified and made independent from those of the project organization.

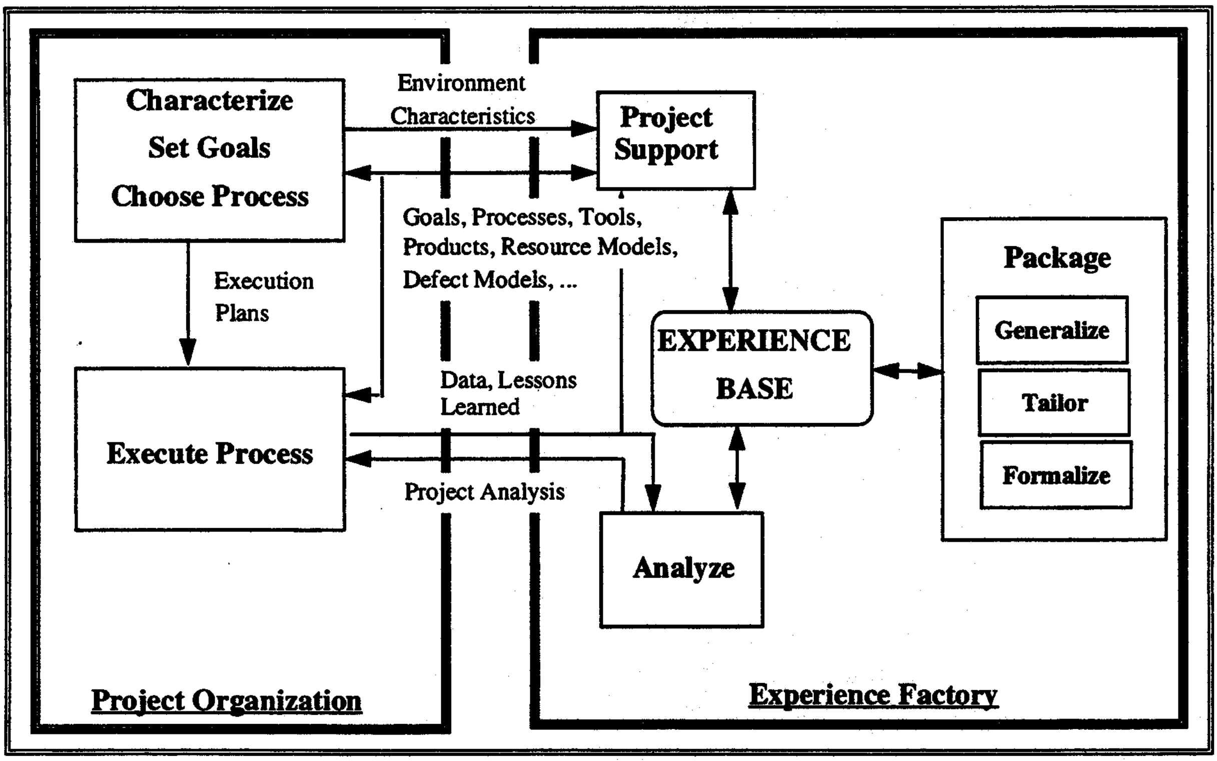
As we have seen at the beginning of this paper, the packaging of experience is based on tenets and techniques that are different from the problem solving activity used in project development. Therefore the projects and the factory will have different process models: each project will choose its process model based upon the characteristics of the software product that will be delivered, while the experience factory will define (and change) its process model based upon the nature of the work, and organizational and performance issues.

Figure 9 provides a high-level picture of the experience factory organization and highlights activities and information flows among the component sub-organizations.

The project organization, whose goal is to produce and maintain software, 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 processed by the experience factory and utilized by the project.

The experience factory provides direct feedback to each project, together with goals and models tailored from similar projects. It also produces and provides upon request baselines, tools, lessons learned, and data, parametrized in some form in order to be adapted to the specific characteristics of a project. The 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 the access to it.

Figure 9



The main product of the experience factory is a set of core competencies packaged as aggregates of technologies. Figure 10 shows some examples of core competencies and the corresponding aggregation of technologies:

Core competencies can be implemented in a variety of formats. We call these formats "experience packages". Their content and structure vary based upon the kind of experience clustered in it. There is, generally, a central element that determines what the package is: a software life cycle product or process, a mathematical relationship, an empirical or theoretical model, a data base, etc. We can use this central element as identifier of the experience package and produce a taxonomy of experience packages based upon the characteristics of this central element; e.g.:

- Product packages: Programs, Architectures, Designs;

Figure 10

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

- Tool packages: Constructive and Analytic Tools;

- Process packages: Process Models, Methods;

- Relationship packages: Cost and Defect Models, Resource Models, etc.;

- Management packages: Guidelines, Decision Support Models;

- Data packages: Defined and validated data, Standardized data, etc.

The operation of the two components is based on the Quality Improvement Paradigm introduced in the previous section. Each component performs activities in all six steps, but for each step one component has a leadership role.

In the first three phases (Characterize, Set Goals, and Choose Process) the focus of the operation is on planning, therefore the project organization has a leading role and is supported by the analysts of the experience factory. 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 phases and the activities of the project, with their products, mutual dependencies, milestones and resources. As far as the experience factory side is concerned, the plan describes the support that the experience factory will provide for each phase and activity, also with products, mutual dependencies, milestones and resources. The two parts of the plan are obviously integrated although executed by different components. The management control frameworks are composed of data (metrics) and models for monitoring the execution of the plan.

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

In the fifth and the sixth phases (Analyze and Package) the focus of the operation is on capturing project experience and making it available to future similar projects, therefore the experience factory has a leading role and is supported by the project organization that is the repository 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 maturity of the organization and incorporate new technologies into the life cycle. In the long term, it supports the overall evolution of the organization from a project-based one, where all activities are aimed at the successful execution of current project tasks, to a capability-based one, which executes those tasks and capitalizes on their execution.

Some important benefits that an organization derives from structuring itself as an experience factory are

- To establish an improvement process for software substantiated and controlled by quantitative data;
- To produce a repository of software data and models which are empirically based on the everyday practice of the organization;
- To develop an internal support organization that represents a limited overhead and provides substantial cost and quality performance benefits;
- To provide a mechanism for identifying, assessing and incorporating into the process, new technologies that have proven to be valuable in similar contexts;
- To incorporate reuse into the software development process and support it;
- To approach in a more software specific way a Total Quality Management program.

The concept of experience factory is an extension and a redefinition of the concept of software factory, as it has evolved from the original meaning of integrated environment to the one of flexible software manufacturing environment [Cusumano, 1991]. The major difference is that, while the software factory is thought of as an independent unit producing code by using an integrated development environment, the experience factory handles all kind of software-related experience. The software factory can be seen as a part of the experience factory, recognizing in this way that its potential benefits can be fully exploited only within this framework.

5. IMPROVEMENT IN PRACTICE: THE NASA SOFTWARE ENGINEERING LABORATORY

In this section we will present and discuss a practical example of experience factory organization. We will show how its operation is based on the Quality Improvement Paradigm and we will use the case of a specific technology in order to illustrate the execution of the steps of the paradigm.

The organization that provides the example is the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center. The laboratory 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 the Computer Sciences Corporation (CSC). The goal of the SEL was to understand and improve key software development processes and products within a specific organization, the Flight Dynamics Division.

In general, the goals, the structure and the 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. At the same time, the awareness of the quality improvement process used in the laboratory has generated the operational paradigm described in this paper as Quality Improvement Paradigm. Today the SEL represents a practical and operational example of experience factory [Basili et al., 1992].

The current structure of the SEL 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 these objects provided by the developers into reusable units and supply them back to the developers; they provide specific support to the projects on the use of the analyzed and synthesized information, tailoring it to a format which is usable by and useful to a current software effort (the Experience Factory proper);

- *Support infrastructure,* which provides services to the developers, on one hand, by supporting data collection and retrieval, and to the analysts, on the other hand, by managing the library of stored information and its catalogs (the Experience Base Support).

The activities of these three sub-organizations, although not separated and independent from each other, have their own goal and process models and plans. Figure 11 outlines the difference in focus among the three sub-organizations.

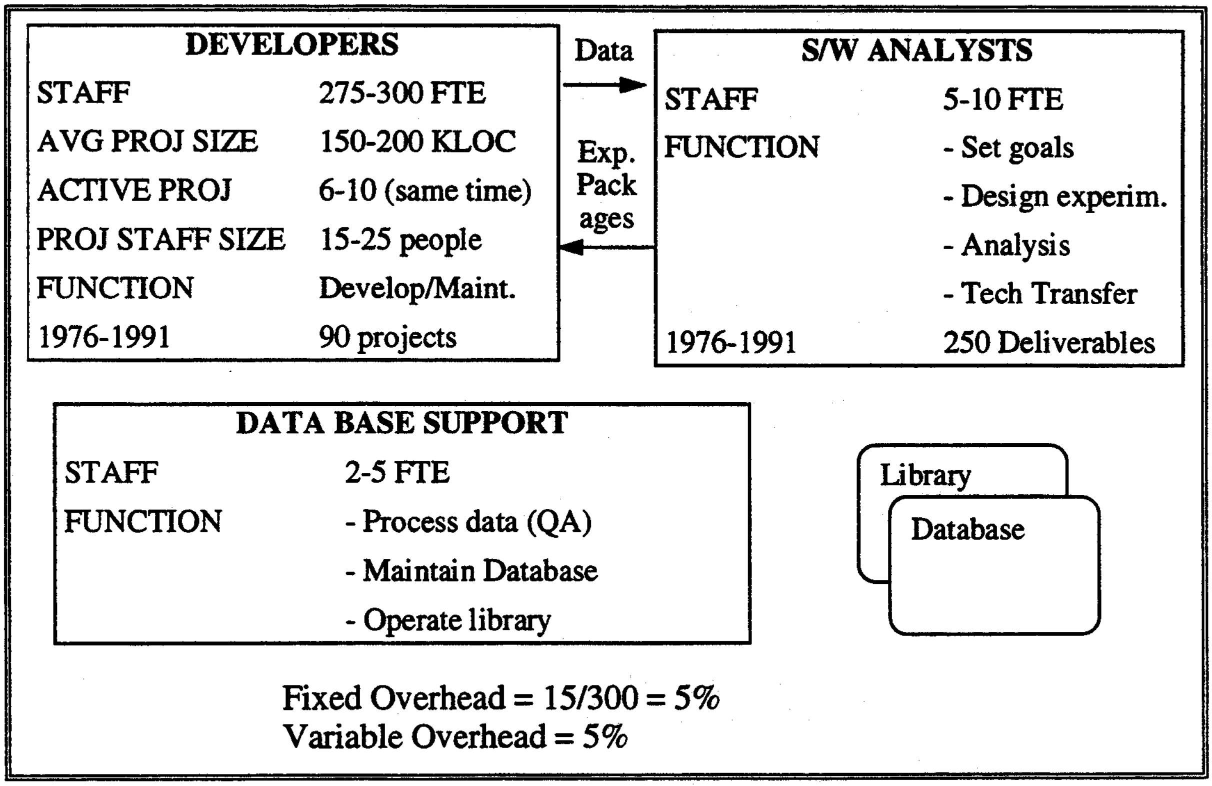
Figure 11

DEVELOPERS FOCUS	ANALYSTS FOCUS	SUPPORT INFRASTRUCTURE 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

Figure 12 gives an idea of the overall size of the organization and of it components.

We will now show the operation of the SEL following the development of a particular core competence through the six steps of the improvement paradigm.

Figure 12



DEVELOPERS	
STAFF	275-300 FTE
AVG PROJ SIZE	150-200 KLOC
ACTIVE PROJ	6-10 (same time)
PROJ STAFF SIZE	15-25 people
FUNCTION	Develop/Maint.
1976-1991	90 projects

S/W ANALYSTS	
STAFF	5-10 FTE
FUNCTION	- Set goals - Design experim. - Analysis - Tech Transfer
1976-1991	250 Deliverables

DATA BASE SUPPORT	
STAFF	2-5 FTE
FUNCTION	- Process data (QA) - Maintain Database - Operate library

Fixed Overhead = 15/300 = 5%
Variable Overhead = 5%

In the late 80's the software engineering community, within and outside NASA, was discussing, among other technologies, the Ada programming language environment and technology [Ada, 1983]: the language had been developed under a major effort of the US Department of Defense and its application was being considered also in areas outside DoD. NASA was, at that time, considering the use of the Ada technology in some major projects such as the Space Station. More and more systems would have used Ada as development environment, and many organizations would have to be involved with it. In consideration of this fact Ada had to be transformed from simple technology to core competence for the software development organizations within NASA.

Associated with Ada there was the issue of object-oriented technologies. It is not very important for our discussion that our reader knows what is an object-oriented design technique. Anyway, Figure 13 provides some basic characteristic elements [Sommerville, 1992] of the object-oriented approach.

Figure 13

Characteristics of the Object-Oriented Approach

- A system is seen as a set of objects having at each time a specific state and behavior
- 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 within the object itself and cannot be deliberately or accidentally used by other objects

The Ada language environment implements several of those features and can be, to a certain extent, considered object-oriented. The design of systems to be implemented in Ada definitely takes advantage of the concepts of object-oriented design. Therefore, from the beginning, there was the impression in the SEL that the two technologies should be packaged together into a core competence supporting the strategic capability of delivering systems with better quality and lower delivery cost. After recognizing 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 supporting data to the decision of using the technology.

The process followed is illustrated in the following steps according to the QIP:

1. Characterize: In 1985, the SEL had achieved a good understanding of how software was developed in the Flight Dynamics Division. The development processes had been defined and models had been built in order to improve the manageability of the process. The standard development methodology, based on the traditional design and build approach, had been integrated 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 that support new languages, like C++ and Ada, offered potential for major improvements in the areas of productivity, quality and reusability of software products and processes, the SEL decided to develop a core competence around object-oriented

design and the use of the programming language Ada. The first step was to set up expectations and goals against which results would be measured. The SEL well-established baseline and set of measures provided an excellent basis for comparison. Expectations included

- A change in the effort distribution of development activities: an increase of the effort on early phases, e.g., design, and a decrease of the effort on late phases, e.g., testing;
- Increased reuse of software modules, both verbatim and with modification;
- Decreased maintenance costs due to the better quality of 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. Two version of the same system would be developed

   System A: To be developed using FORTRAN and following the standard methodology based on functional decomposition. This system will become operational and its development will follow the ordinary schedule constraints.

   System B: To be developed using Ada and following an object-oriented methodology called OOD. This system will not become operational.

The data derived from the development of System B would be compared with those derived from the development of System A. Particular attention would be dedicated to quality and productivity data. The data collection and comparison would be based on the Goal Question Metric Model shown in Figure 14.

Figure 14

Goal	Purpose	Evaluate the impact of
	Object	the object-oriented approach and Ada
	Issue	on the quality and productivity
	Viewpoint	within the Flight Dynamics Division
Question	1	What is the impact on the cost to develop software?
Metrics	1.1	Number of hours per statement developed for System A
	1.2	Number of hours per statement developed for System B
Question	2	What is the impact on the cost to deliver software?
Metrics	2.1	Number of hours per statement included in System A
	2.2	Number of hours per statement included in System B
Question	3	What is the impact on the quality of the delivered software?
Metrics	3.1	Number of defects per 1000 lines of code in System A
	3.2	Number of defects per 1000 lines of code in System B
Question	4	What was the amount of reuse that occurred?
Metrics	4.1	Percentage of reused code

4. Execute: System A and B were implemented and the desired metrics were collected. During the development changes had to be applied to the approach that was used for using Ada and also adaptations had to be made in order to use OOD. For instance: some review procedures that were particularly suited for a design based on functional decomposition did not fit the approach used for System B. Therefore new review procedures were drafted for that development.

5. Analyze: The data collected based on the previous GQM model showed an increase of the cost to develop (Metrics 1.1 and 1.2) that was interpreted as due on one hand to the inexperience of the organization with the new technology and on the other hand to the intrinsic characteristics of the technology itself. The data also showed an increase in the cost to deliver (Metrics 2.1 and 2.2) interpreted as due to the same

causes. The overall quality of System B showed an improvement over System A (Metrics 3.2 and 3.1) in terms of a substantially lower error density. Reuse data across systems (Metric 4.1) were obviously not available for System B because of the new implementation technology. The comparative data are shown in Figure 15.

Figure 15

Measure	System A	System B
Cost to develop (Hrs per Stm)	0.70	1.00
Cost to deliver (Hrs per Stm)	0.65	1.00
Defect density (Def. per 1000 lines of code)	3.90	1.80
Reuse (%)	30%	N/A

6. Package: The laboratory tailored and packaged an internal version of the methodology which adjusted and extended OOD 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 containing the lessons learned using the new technology and recommending refinements to the methodology and the standards.

The data collected from the first execution of the process were encouraging, especially on the quality issue, but not conclusive. Therefore new executions were decided and carried over in the following years. In conjunction with the development methodology, a programming language style guide was developed, that provided coding standards for the local Ada environment. At least 10 projects have been completed by the SEL 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 thousand lines of source code (KSLOC), ranges from small (38 KSLOC) to large (185 KSLOC). 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% reuse across different projects, to a current 96% (89% verbatim reuse).

Over the years the use of the object-oriented approach and the expertise with Ada have matured. Source code analysis of the systems developed with the new technology has revealed a maturing use of key features of Ada that have no equivalent in the programming environments traditionally used at NASA. Such features were not only used more often in more recent systems, but they were also used in more sophisticated ways, as revealed by specific metrics used to this purpose. Moreover, the use of object-oriented design and Ada features has stabilized over the last 3 years, creating an SEL baseline for object-oriented developments.

The charts shown in Figure 16 represent the trend of some significant indicators.

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 lies now well below the old cost to deliver.

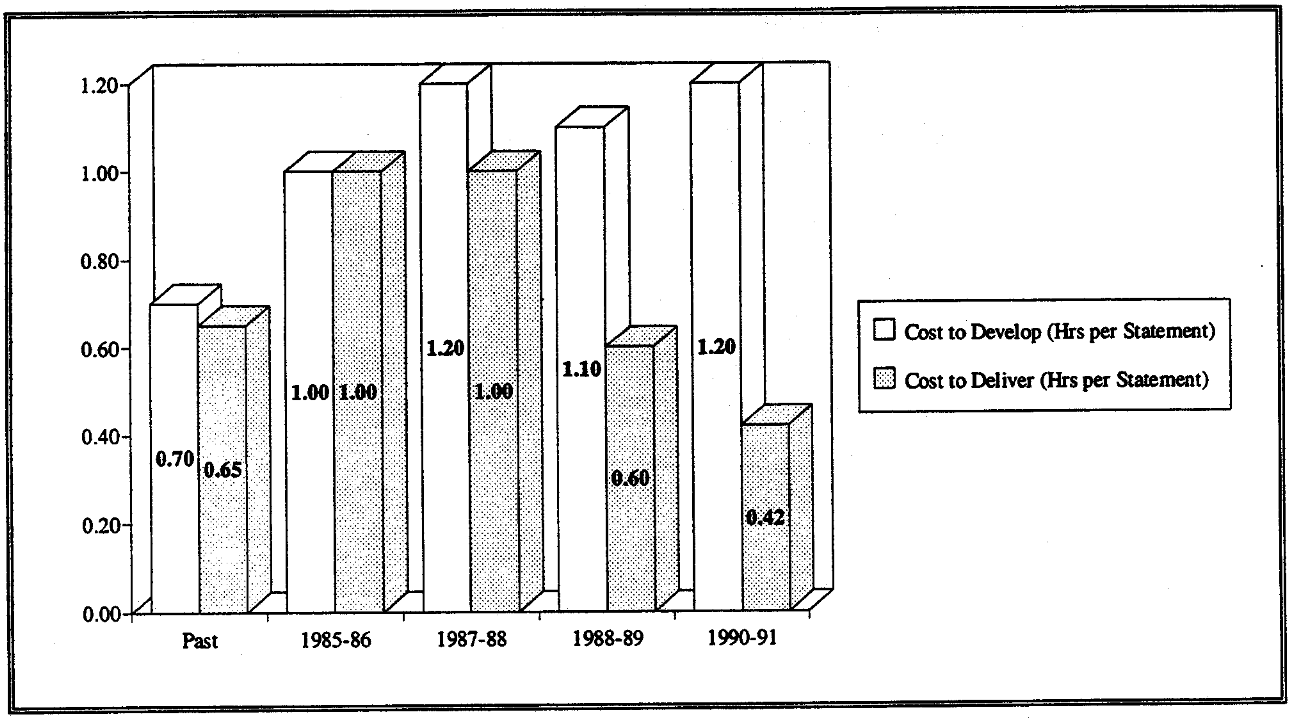
The reliability of the systems developed in the new environment has improved over the years with the maturing of the technology. Although the error rates were significantly lower than the traditional ones, they have continued to decrease even further: again, the high level of reuse in the later systems is a major contributor to this greatly improved reliability.

Because of the stabilization of the technology and apparent benefit to the organization, the object-oriented development methodology has been packaged and incorporated into the current technology baseline and is a core competence of the organization. And this is where things stand today.

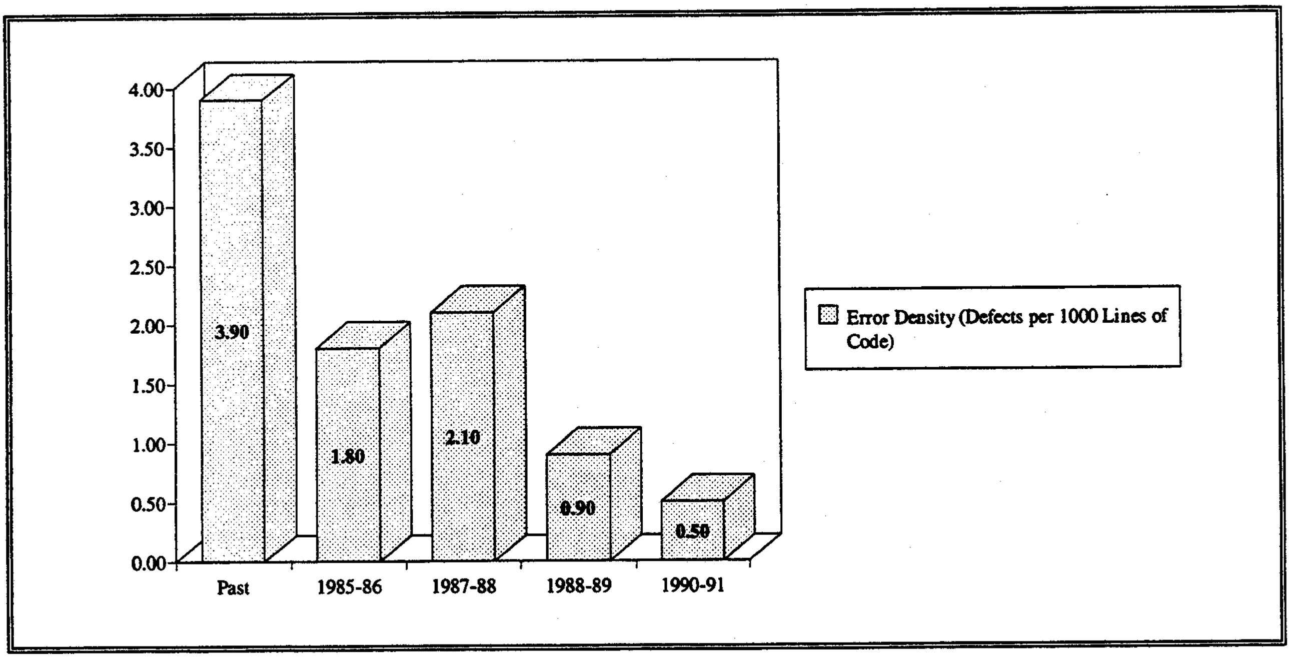
Although the technology of object-oriented design will continue to be refined within the SEL, it has now progressed through all stages, moving from a candidate trial methodology to a fully integrated and packaged part of the standard methodology, ready for further incremental improvement.

The example we have just shown illustrates also the relationship between a competence (object-oriented technology) and a target capability (deliver high quality at low cost), and shows how innovative technologies can enter the production cycle of mature organizations in a systematic way. Although the topic of technology transfer is not within the scope of this paper, it is clear from the SEL example that the model we derive from it outlines a solution to some major technology transfer issues.

Figure 16



1.20
1.00
0.80
0.60
0.40
0.20
0.00
0.70
0.65
1.00
1.00
1.20
1.00
1.10
0.60
1.20
0.42
Past
1985-86
1987-88
1988-89
1990-91
□ Cost to Develop (Hrs per Statement)
▨ Cost to Deliver (Hrs per Statement)



4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
3.90
1.80
2.10
0.90
0.50
Past
1985-86
1987-88
1988-89
1990-91
▨ Error Density (Defects per 1000 Lines of Code)

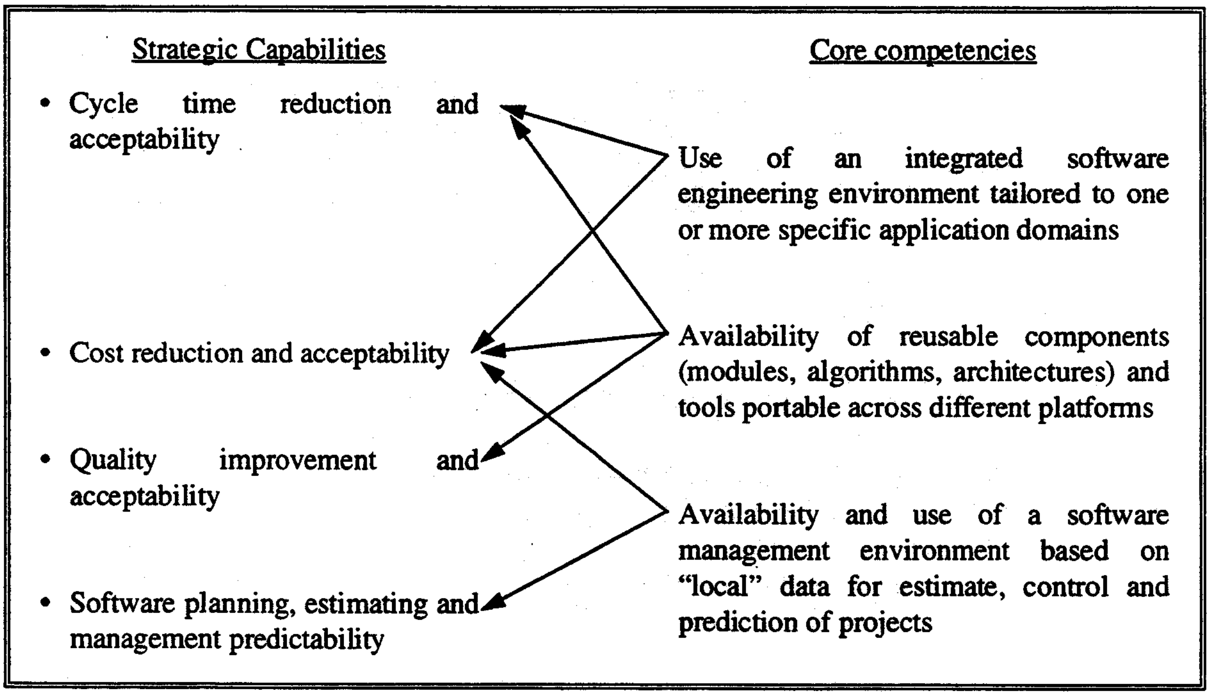
The purpose of an experience factory organization is larger than technology transfer: it is capability transfer and reuse. If these capabilities are already consolidated into a technology, available within the organization or outside it, then the process is a process of technology transfer. If the capabilities are present in the organization as informal experience, products prepared for other purposes, and lessons learned, then the process is different from technology transfer.

6. CONCLUSIONS

Clearly the nineties will be the quality era for software and there is a growing need to develop or adapt quality improvement approaches to the software business. Our approach to software quality improvement, as it has been presented in this paper, is based on the exploitation and reuse of the critical capabilities of an organization 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. A possible mapping is shown as an example in Figure 17, in the case of an organization whose main business is development of systems and software for user applications.

Figure 17


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

In this paper we have shown, through the NASA example, that all these ideas are practically feasible and have been successfully applied in a production environment in order to create a continuously improving organization.

But what does "continuously improving organization" really mean? It is an organization that can manipulate its processes to achieve various product characteristics. This requires that the organization has a process and an 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.

Using the Quality Improvement Paradigm/Experience Factory Organization approach the organization has a good chance to achieve all these capabilities, and to move up in the quality excellence scale 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.

ACKNOWLEDGMENTS

The authors would like to acknowledge the contributions given to this paper by all those who participated in the experiences and discussions that originated the concepts presented in this paper. The list of their names would be too long. A particular acknowledgment goes to the personnel of the Software Engineering Laboratory at NASA Goddard Space Flight Center and Frank McGarry who made it possible. Other acknowledgments go to Jerry Page (CSC), Tony Jordano (SAIC), Bob Yacobellis (Motorola), Paolo Sigillo (Italsiel).

REFERENCES

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

[Basili and Weiss, 1984]
V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering,* November 1984, pp. 728-738.

[Basili, 1985]
V. R. Basili, "Quantitative Evaluation of a Software Engineering Methodology", *Proceedings of the First Pan Pacific Computer Conference,* Melbourne, Australia, September 1985.

[Basili and Rombach, 1988]
V. R. Basili, H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Transactions on Software Engineering,* June 1988, pp. 758-773.

[Basili, 1989]
V. R. Basili, "Software Development: A Paradigm for the Future (Keynote Address)", *Proceedings COMPSAC '89,* Orlando, FL, September 1989, pp. 471-485.

[Basili, Caldiera, and Cantone, 1992]
V. R. Basili, G. Caldiera and G. Cantone, "A Reference Architecture for the Component Factory", *ACM Transactions on Software Engineering and Methodology,* Vol. 1, No. 1, January 1992, pp. 53-80.

[Basili, Caldiera, McGarry, Pajerski, Page, and Waligora, 1992]
V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, J. Page, and S. Waligora, "The Software Engineering Laboratory - An Operational Software Experience Factory", *Proceedings of the Fourteenth International Conference on Software Engineering,* Melbourne, Australia, May 1992.

[Bootstrap]
2I Industrial Informatics, *BOOTSTRAP Project Proposal and Mission Statement,* 2I GmbH, Haierweg 20e, D7800 Freiburg, Germany, 1990, 1991

[Cusumano, 1991]

M.A. Cusumano, *Japan's Software Factories*, Oxford University Press, New York, 1991.

[Deming, 1986]
W. Edwards Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, MIT Press, Cambridge, MA, 1986.

[Dewan and Riedl, 1993]
P. Dewan and J.Riedl, "Toward Computer-Supported Concurrent Software Engineering", *IEEE Computer, Special issue on Computer Support for Concurrent Engineering*, January 1993, pp. 17-27.

[Dion, 1993]
R. Dion, "Process Improvement and the Corporate Balance Sheet", *IEEE Software*, July 1993, pp. 28-35.

[Feigenbaum, 1991]
A. V.. Feigenbaum, *Total Quality Control*, Fortieth Anniversary Edition, Mc Graw Hill, New York, NY, 1991.

[Hamel and Prahalad, 1990]
G. Hamel, C. K. Prahalad, The Core Competence of the Corporation, *Harvard Business Review*, Vol. ?, No. ?, July-August 1991, pp. 79-??.

[Hamel and Prahalad, 1991]
G. Hamel, C. K. Prahalad, Corporate Imagination and Expeditionary Marketing, *Harvard Business Review*, Vol. 69, No. 4, July-August 1991, pp. 81-92.

[ISO1]
ISO 8402: 1986 *Quality - Vocabulary*

[ISO2]

ISO 9000: 1987 *Quality Management and Quality Assurance Standards - Guidelines for Selection and Use*

ISO 9001: 1987 *Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing*

ISO 9001-3: 1991 *Quality Management and Quality Assurance Standards - Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*

[ISO3]

ISO 9126: 1991 *Software Product Evaluation - Quality Characteristics and Guidelines for their Use*

[SEI]

W. S. Humphrey, W. L. Sweet, *A Method for Assessing the Software Engineering Capability of Contractors*, Software Engineering Institute, Technical Report, CMU/SEI-87-TR-23, September 1987.

M. C. Paulk, B. Curtis, M. B. Chrissis, *Capability Maturity Model for Software*, Software Engineering Institute, Technical Report, CMU/SEI-91-TR-24, August 1991.

[Stalk, Evans, and Shulman, 1992]

G. Stalk, P. Evans, and L. E. Shulman, "Competing on Capabilities: The New Rules of Corporate Strategy", *Harvard Business Review*, Vol. 70, No. 2, March-April 1992, pp. 57-69.

[Sommerville, 1992]

Ian Sommerville, *Software Engineering*, Fourth Edition, Addison-Wesley, Wokingham, England, 1992.

[Womack, 1989]

J. P. Womack, D. T. Jones, D. Roos, D. S. Carpenter, *The Machine that Changed the World*, Rawson Associates (MIT Study on Lean Production), New York, NY, 1989.