Software quality: An overview from the perspective of total quality management

by S. H. Kan
V. R. Basili
L. N. Shapiro

This essay presents a tutorial that discusses software quality in the context of total quality management (TQM). Beginning with a historical perspective of software engineering, the tutorial examines the definition of software quality and discusses TQM as a management philosophy along with its key elements: customer focus, process improvement, the human side of quality, and data, measurement, and analysis. It then focuses on the software-development specifics and the advancements made on many fronts that are related to each of the TQM elements. In conclusion, key directions for software quality improvements are summarized.

From a historical perspective, the 1960s and the years prior to that decade could be viewed as the functional era of software engineering, the 1970s as the schedule era, and the 1980s as the cost era.[1] In the 1960s, we learned how to exploit information technology to meet institutional needs, and began to link software with the daily operations of institutions in society. In the 1970s, when software development was characterized by massive schedule delays and cost overruns, the focus was on planning and control of software projects. Phase-based life-cycle models were introduced, and analyses like the mythical man-month[2] emerged. In the 1980s, hardware costs continued to decline. Information technology permeated every facet of our institutions, and at the same time it became available to individuals. As competition in the computer industry became keen and low-cost applications became widely implemented, the importance of productivity in software development increased significantly. Various cost models in software engineering were developed and used. In the late 1980s, the importance of quality was also recognized.

The 1990s and beyond is certainly the quality era. As state-of-the-art technology is now able to provide abundant functionality, customers demand high quality. Demand for quality is further intensified by the ever-increasing dependence of our society on software. Billing errors, large-scale disruptions of telephone services, and even a missile failure during the recent Gulf War[3] can all be traced to the issue of software quality. In this era, quality has been brought to the center of the software development process. From the standpoint of software vendors, quality has become a necessary condition to compete in the marketplace.

This essay provides a high-level tutorial on software quality and total quality management (TQM). In the following sections, we discuss the defini-

tion of software quality, the TQM philosophy, and the progress on many fronts of software engineering as they relate to the key TQM elements: customer focus, process improvement, the human side of quality, and data, measurement, and analysis. Because the subject is very broad, in this tutorial we take a concise, high-level approach. For in-depth information related to specific topics, the reader is encouraged to peruse the references.

Meaning and definition of software quality

Quality must be defined and measured for improvement to be achieved. Yet, a major problem in quality engineering is that the word quality lacks a commonly recognized operational definition. Perhaps the confusion is because quality is not a single idea, but a multidimensional concept. The dimensions of quality include the entity of interest, the viewpoint on that entity, and the quality attributes of that entity. A popular view of quality is that it is an intangible trait and that it can be discussed and judged, but cannot be weighed or measured. To many people, quality is similar to what a federal judge once observed about obscenity: "I know it when I see it." From a customer's standpoint, quality is the customer's perceived value of the product he or she purchased, based on a number of variables, such as price, performance, reliability, overall satisfaction, and others. In Guaspari's book *I Know It When I See It*,[4] the author discusses quality in the customers' context as follows:

> Your customers are in a perfect position to tell you about Quality, because that's all they're really buying. They're not buying a product. They're buying your assurances that their expectations for that product will be met.
>
> And you haven't really got anything else to sell them but those assurances. You haven't really got anything else to sell but Quality.

It is clear that the concept of quality must involve customers or, simply put, quality is conformance to customers' expectations and requirements. Interestingly, the definitions of quality by quality professionals are congruent with the implications of the popular views. For instance, Crosby's "conformance to requirements"[5] and Juran's "fitness for use"[6] both implied the customers' perspective.

From a high-level definition of a concept, to a product being operationally defined, many steps are involved, each of which may be exposed to possible shortcomings. For example, to achieve the state of conformance to requirements, customers' requirements must first be gathered and

Quality is conformance to customers' expectations and requirements.

analyzed, then specifications from those requirements must be developed, and the product must be developed and manufactured appropriately. In each phase of the process, errors may have occurred that will negatively affect the quality of the finished product. The requirements may be erroneous (especially in the case of software development), the development and manufacturing process may be subject to variables that induce defects, and so forth. From the customer's perspective, satisfaction after the purchase of the product is the ultimate validation that the product conforms to requirements and is fit to use. From the producer's perspective, once requirements are specified, developing and producing the product in accordance with the specifications is the basic step to achieving quality. Usually, for product quality, lack of functional defects and good reliability are the most basic measures. In order to be "fit for use," the product first has to be reliably functional.

Because of the two perspectives on quality (i.e., customer satisfaction as the ultimate validation of quality, and the producer's adherence to requirements to achieve quality), the *de facto* definition of quality consists of two levels. The first is the intrinsic product quality, often operationally limited to product defect rate and reliability; this narrow definition is referred to as the "small" q (q for quality). The broader level of the definition of quality includes both product quality and customer satisfaction; it is referred to as the "big" Q. One can observe that this two-level approach to

quality is being used in many industries, including the automobile industry, the computer industry (both software and hardware), the consumer electronics industry, and many others.

In software, the narrowest sense of product quality is commonly recognized as lack of "bugs" in the product. This definition is usually expressed in two ways: defect rate (e.g., number of defects per million lines of source code, or per function point), and reliability (e.g., number of failures per n hours of operation, mean time to failure, or the probability of failure-free operation in a specified time). Customer satisfaction is usually measured by the percentage of those satisfied or nonsatisfied (neutral and dissatisfied) on customer satisfaction surveys. To reduce bias, usually techniques such as double-blind surveys (the interviewer not knowing who the customer is, and the customer not knowing what company the interviewer represents) are used. In addition to overall customer satisfaction with the software product, satisfaction toward specific attributes is also gauged. For instance, IBM monitors the CUPRIMDSO satisfaction levels of its software products (i.e., capability [functionality], usability, performance, reliability, installability, maintainability, documentation/information, service, and overall satisfaction). The Hewlett-Packard Co. focuses on FURPS (functionality, usability, reliability, performance, and supportability).[7] Similar dimensions of software customer satisfaction are used by other companies.

The two-level concept of quality is supposed to form a closed-loop cycle: customers' wants and needs → requirements and specifications → products designed, developed, and manufactured in accordance with the requirements → excellent product quality plus good distribution and service processes → total customer satisfaction. However, this concept had not always been present in many industries, especially before the late 1980s when the modern quality era began. Product requirements were often generated without customers' input, and customer satisfaction was not always a factor in business decision-making. Although the final products conformed to requirements, they may not have been what the customers wanted. Therefore, we think that the role of customers should be explicitly spelled out in the definition of quality: conformance to customers' requirements. This updated definition is especially relevant to the software industry, since requirements errors constitute one of the major problems of that industry, specifically about 15 percent of defect totals.[8] A development process that does not address requirements quality is bound to produce poor quality software.

Yet another view of software quality is that of process quality versus end-product quality. From requirements to the delivery of software products, the development process is complex. It often involves a series of stages, each with feedback paths. In each stage, an intermediate deliverable is produced for an intermediate user—the next stage. Each stage also receives an intermediate deliverable from the preceding stage. Each intermediate deliverable has certain quality attributes that affect the quality of the end product. Intriguingly, if we extend the concept of customer in the definition of quality to include both external and internal customers, the definition also applies to process quality. If each stage of the development process meets the requirements of its intermediate user (the next stage), the end product thus developed and produced will meet the specified requirements. This statement, of course, is an oversimplification of reality, because in each stage numerous factors exist that will affect the ability of the stage to fulfill its requirements completely. However, if each person performing an activity thought about the customers of the intermediate product being developed, and applied the concepts discussed above, we would come a long way toward improving the final product quality.

Another view of process quality is aimed at improving the processes themselves, i.e., defining a set of ideal processes and measuring the existing processes of organizations against these ideals. This concept has become very popular in the last decade and provides a mechanism for companies to be related with regard to process.[9]

To improve quality during development, we need models of the development process, and within the process we need to select and deploy specific methods and approaches and employ proper tools and technologies. We need measures of the characteristics and quality parameters of the development process and its stages. We need metrics and quality models to help ensure that the development process is under control to meet the quality objectives of the product.

Total quality management

Total quality management (TQM) is a term that was originally coined in 1985 by the Naval Air Systems Command to describe its Japanese-style management approach to quality improvement. It has taken on a number of meanings, depending on who is interpreting the phrase and how they are applying it, but in general, it represents a style of management that is aimed at achieving long-term success by linking quality with customer satisfaction. Basic to the approach is the creation of a culture in which all members of the organization participate in the improvement of processes, products, and services. Various specific methods for implementing the TQM philosophy are found in the works of Philip Crosby,[5] W. Edwards Deming,[10] Armand V. Feigenbaum,[11,12] Koru Ishikawa,[13] and J. M. Juran.[6]

To improve quality during development, we need models of the development process.

Since the 1980s, many companies in the United States have begun adopting the TQM approach to quality. The Malcolm Baldrige National Quality Award (MBNQA), established by the U.S. government in 1988, highlighted the embracement of such a philosophy and management style. In the computer and electronic industry, examples of successful implementation include Hewlett-Packard Co.'s total quality control (TQC), Motorola Inc.'s six sigma strategy, IBM's market-driven quality, and others. In fact, Motorola won the first MBNQA award in 1988, and IBM's AS/400 Division in Rochester, Minnesota, was one of the winners in 1990.

Hewlett-Packard's TQC focuses on key areas such as management commitment, leadership, customer focus, total participation, and systematic analysis. There are strategies and plans in each area to drive the improvement of quality, efficiency, and responsiveness. The final objective is to achieve success through customer satisfaction.[14] In software development, the Software Quality and Productivity Analysis (SQPA) program[15] is one of the approaches taken to improve quality.

Motorola's six sigma strategy focuses on achieving stringent quality levels to obtain total customer satisfaction. Cycle time reduction and participative management are among the key initiatives of the strategy.[16] Six sigma is not just a measure of the quality level; inherent in the concept are product design improvements and reductions in process variations.[17] Six sigma is applied to product quality as well as to everything that can be supported by data and measurement.

"The customer is the final arbiter" is the key theme of IBM's market-driven quality strategy. The strategy consists of four initiatives: defect elimination, cycle time reduction, customer and Business Partner satisfaction, and adherence to the Baldrige assessment discipline.

Despite variations in its implementation, the key elements of a TQM system can be summarized as follows:

- Customer focus: the objective is to achieve total customer satisfaction. Customer focus includes studying customers' wants and needs, gathering customer requirements, and measuring and managing customer satisfaction.
- Process: the objective is to reduce process variations and to achieve continuous process improvement. Process includes both the business process and the product development process. Through process improvement, product quality will be enhanced.
- Human side of quality: the objective is to create a company-wide quality culture. Focus areas include management commitment, total participation, employee empowerment, and other social, psychological, and human factors.
- Measurement and analysis: the objective is to drive continuous improvement in all quality parameters by the goal-oriented measurement system.

A variety of organizational frameworks that have been proposed to improve quality can be used to substantiate the TQM philosophy. Specific examples include Plan-Do-Check-Act,[10,18] Quality Improvement Paradigm/Experience Factory Orga-

nization,[19-23] the Software Engineering Institute (SEI) Capability Maturity Model,[9,24] and Lean Enterprise Management.[25]

Plan-Do-Check-Act is a quality improvement process based on a feedback cycle for optimizing a single process or production line. It uses such techniques as feedback loops and statistical quality control to experiment with methods for improvement and to build predictive models of the product. A basic assumption is that a process is repeated multiple times so that data models can be built that allow one to predict results of the process.

To achieve good quality all TQM elements need to be focused.

The Quality Improvement Paradigm (QIP)/Experience Factory (EF) Organization aims at building a continually improving organization based on its evolving goals and an assessment of its status relative to those goals. The approach uses internal assessments against the organization's own goals and status (rather than process areas) and such techniques as Goal/Question/Metric (GQM), model building, and qualitative/quantitative analysis to improve the product through the process. The six fundamental steps of the QIP are (1) characterize the project and its environment, (2) set the goals, (3) choose the appropriate processes, (4) execute the processes, (5) analyze the data, and (6) package the experience for reuse. The Experience Factory Organization separates the product development from the experience packaging activities. A main element of the QIP/EF is the need to learn across multiple project developments.

The SEI Capability Maturity Model is a staged process improvement based on assessment with regard to a set of key process areas until a level 5 is reached, which represents a continuous process improvement. The approach is based on organizational and quality management maturity models developed by Likert[26] and Crosby,[5] respectively. The goal of the approach is to achieve continuous process improvement via defect prevention, technology innovation, and process change management.

As part of the approach, a five-level process maturity model is defined based on repeated assessments of the capability of an organization in key process areas. Improvement is achieved by action plans for poor areas. Basic to this approach is the idea that there are key process areas that will improve software development.

Lean Enterprise Management is based on the principle of concentration of production on "value added" activities and the elimination or reduction of "not value added" activities. The approach has been used to improve factory output. The goal is to build software using the minimal set of activities needed and tailoring the process to the product needs. The approach uses such concepts as technology management, human-centered management, decentralized organization, quality management, supplier and customer integration, and internationalization/regionalization. A key point of this approach is that the process can be tailored to classes of problems.

It is not surprising that our discussion of the definition of software quality in the previous section fits perfectly in the TQM context. That definition correlates closely with the first two of the TQM elements listed above. And to achieve good quality, definitely all TQM elements need to be focused, with the aid of some organizational frameworks. In the following sections, we discuss software specifics in terms of the TQM elements. We first discuss the customer focus activities in software development. Then we discuss: process improvements and technology advancements as they relate to development quality; the human factors that are relevant to software engineering; and the role of data, measurement, and models in software development.

Customer focus in software development

As discussed earlier, requirements defects constitute a large portion of software defects. Gauging customers' wants and needs and gathering customers' requirements have become increasingly important in the software industry. Prac-

tices like customer advisory councils, formal requirements gathering processes, and development of line items to address specific dimensions of customer satisfaction have become common

When customer requirements are verified, the challenge is to reflect them in the product quality development process.

among large developers. To verify requirements with customers, fast feedback is often needed. To accomplish this task, tools and methods in quality engineering, in combination with computer-aided software engineering (CASE) technology, could prove to be very beneficial. For instance, a large software developer implemented the seven new quality management tools (the Affinity Diagram, the Interrelationship Diagram, the Tree Diagram, the Matrix Chart, the Matrix Data Analysis Chart, the Process Decision Program Chart, and the Arrow Diagram) in the CASE tool set for its customer requirements gathering and verification process, and obtained positive results with significant cycle time reduction.[27]

When customer requirements are verified, the challenge is to reflect them in the product development process. The quality function deployment (QFD) approach has been used in other industries. In software, QFD has found its use in application development (for instance, in IBM Japan and some divisions in IBM United States). For system software, the application of QFD in bridging customer requirements with product development may take more ingenuity. The QFD approach was originally developed for manufacturing in order to better define and understand customer requirements and map them into the various characteristics of the product design.[28] QFD uses models and metrics to plan the control and engineering of the product to satisfy customers' needs.

Various software process models have been defined that attempt to deal with customer feedback on the product to ensure that it satisfies the requirements. Each of these models provides some form of prototyping, either of a part or all of the system. Some developers build prototypes to be thrown away, others have the prototype evolve over time, based on customer needs.

The rapid throwaway prototyping approach of software development, made popular by Gomaa,[29] is now widely used in the industry, especially in application development. A prototype is a partial implementation of the product, expressed either logically or physically, with all external interfaces presented. Potential customers use the prototype and provide feedback to the development team before full-scale development starts. An old saying is, "Seeing is believing," and that is really what prototyping intends to achieve. Using this approach, the customers and the development team can clarify both their requirements and their interpretation. An extension of this approach uses a series of throwaway prototypes that escalate into full-scale development. Such a process is known as the spiral model.[30]

The iterative enhancement (IE) approach[31] was defined to begin with a subset of the requirements and to develop a subset of the product that satisfies the essential needs of the users, provides a vehicle for analysis and training for the customers, and provides a learning experience for the developer. With the analysis of each intermediate product as a foundation, the design and the requirements are modified over a series of iterations to provide users with a system that meets evolving customer needs with improved design, based on feedback and learning.

The iterative enhancement approach, or the iterative development process (IDP), combines the strength of the classical waterfall model with prototyping. Other methods, such as domain analysis and risk analysis, can also be incorporated into the process model. For instance, the IDP implemented by IBM Owego contains eight major steps: domain analysis, requirements definition, software architecture, risk analysis, prototype, test suite and environment development, integration with previous iterations, and iteration release.[32] With the purpose of "build a system by evolving an architectural prototype through a series of executable versions, with each successive iteration incorporating experience and more system func-

tionality," IBM Owego's IDP fits well with the IE framework defined by Basili and Turner.[31]

The development of IBM's Operating System/2* (OS/2*) 2.0 is a combination of the iterative development process and the small team approach. Somewhat different from the last example, the OS/2 2.0 iterative development process involved large-scale early customer feedback instead of just prototyping. The iterative part of the process involved the steps of subsystem design, subsystem code and test, system integration, and customer feedback. Within the one-year development cycle, five iterations took place before the final completion of the system, each with increased functionality. A total of more than 30 000 copies of the software was distributed, with over 100 000 users having installed the product during the iteration feedback process. Supporting the iterative process was the small team approach, in which each team assumed full responsibility for a particular function of the system. Each team "owned" its project, functionality, and quality, as well as customer satisfaction, and was held totally responsible.[33] The OS/2 2.0 development process and approach, although not necessarily universally applicable to other products and systems, were apparently a true success, as attested to by customers' acceptance of the product and positive responses.

Customer focus at the back end of the development process, in the form of early customer feedback and customer burn-in programs, is widely adopted regardless of the development process. For example, in the modified waterfall development process used for large-scale projects by major developers, early customer programs are often a well-defined phase in the process. During the phase of early customer programs, the order, distribution, install, and service processes are verified with selected customers. The customers' feedback on product characteristics is also gathered. Process and product problems uncovered in the verification and feedback processes are corrected before the product becomes available to the general market.

Customer burn-in programs are a special type of early customer program, with the specific purpose of achieving further defect reduction and increased reliability of the software product. In traditional quality engineering, customer burn-in is a well-known approach to quality improvement. In software, customer burn-in programs can be used to accelerate the field aging process. Customers with a history of high defect discovery and those who will exercise the new functions are good candidates for the program (when the product development is complete but the product is not yet available to the general market). Usually, extra technical support from the developer is provided to minimize customers' risks, and customers are requested to exercise the functions in their production scenarios in order to flush out latent defects. Defects found by the program are fixed immediately; therefore, quality is improved and reliability grows when the product becomes generally available. The Customer Quality Partnership (CQP) program of IBM's AS/400 Division (IBM Rochester) and the Quality Partnership Program (QPP) of IBM's Software Solutions Division are examples of use of burn-in programs in large-scale software development.

Process, technology, and development quality

Given the customers' requirements, the central question is how to develop the software effectively so that it can meet the criterion of "conformance to customers' requirements." In past years, advancements have been made on many fronts of software engineering, and these advancements are leading to more efficiency and effectiveness, and better quality.

Defect prevention process. The defect prevention process (DPP) is one of the process improvement approaches that originated in software development. It was modeled on similar techniques used in manufacturing for many decades (for example, quality circles) and is in agreement with Deming's principles. The formal process, first used at the IBM Communications Programming Laboratory at Research Triangle Park, North Carolina, consists of four key elements:[34,35]

- Formal causal analysis meetings—These are brainstorming sessions conducted by technical members at the end of each stage of the development process. Members analyze defects that occurred at each stage, trace the root causes, and brainstorm possible actions to prevent similar errors from recurring.
- Action team—Being the engine of the process, the action team is responsible for screening, prioritizing, and implementing suggested actions

from causal analysis meetings. The team is also involved in providing feedback to the organization, reports to management on the status of its activities, publicizing success stories, and taking the lead in various aspects of the process.

- Stage kickoff meetings—These meetings are conducted by the technical members at the beginning of each development stage. They serve as the key preventive measure as well as a primary feedback mechanism of progress made. The emphasis is on the technical aspect of the development process as well as on quality: "What is the right process? How do we do things more effectively? What are the tools and methods that can help? What are the common errors to avoid? What improvements and actions have been implemented?"
- Action tracking and data collection—An action database is needed to track action status, to facilitate implementation, to prevent action items from being lost over time, and to enhance communications among groups.

Different from postmortem analysis, the DPP is a real-time process, integrated into every stage of the development process. Through the action teams and action tracking tools and methodology, DPP provides a systematic, objective, data-based mechanism for action implementation. It injects intelligence into the development process and enables the development process to refine itself. Developed and first used by IBM Research Triangle Park in the mid-1980s, it is now being used by many other IBM development organizations and by other companies in the software industry. DPP can be applied regardless of the type of development process.

Design reviews. Design reviews, code inspection, and code walk-through or code reading are the classical techniques to ensure in-process quality. When used in the waterfall process, they are usually part of the exit criteria for development phases. For instance, when the high-level design is done, a formal review is held; design defects found by the review must be fixed before exiting the phase (to the low-level design phase). Recent improvements in reviews and inspections include: the phased inspection method, in which specific tasks are assigned to specific inspectors and two or more phases of inspection (of different complexity levels) are conducted;[36] various online review tools (for example, REVUFILE, used at the IBM Santa Teresa laboratory); and the use of multimedia technology (for example, the Santa Teresa laboratory's multimedia approach that includes large-screen projector, BARCO machine,

Software reviews and inspections are distinctly different from manufacturing inspections.

Personal System/2* computers, intelligent source code editor, and other tools). Improvements in these techniques will lead to better quality of the front-end development process.

It should be cautioned that software reviews and inspections are distinctly different from manufacturing inspections. The latter are at the back end of the production process and are known to be a poor method for quality assurance. TQM teachings often call for the abandonment of manufacturing inspections in favor of acceptance sampling (with the front-end focus on design quality). Software reviews and inspections, on the contrary, are the vital techniques at the front end of the software development process. To have reviews and inspections by peers on software design and implementation is beneficial. As new languages emerge that can make the mundane task of code implementation a lot simpler (for example, languages with strong typing), the burden of code inspection can be lessened. However, design reviews or inspections will become ever more important, regardless of the development process.

Formal methods. In computation theory, computer scientists have developed models based on mathematical formalisms. These formalisms include, for example, predicate calculus, functional verification, and state machines. Because of the difficulty in scaling up to reasonable-sized systems and the mathematical training required, these models have not been used effectively in practice.

The last several years have seen the initial applications of formal methods in software develop-

ment. Examples include the Vienna Development Method (VDM), Z notation, Input/Output Requirements Language (IORL),[37,38] and the Cleanroom methodology.[39–41] It appears that Z notation and VDM have primarily been used by developers in Europe, whereas Cleanroom projects have taken place mostly in the United States.

The Cleanroom methodology involves box structure specification of user function and system object architecture, function-theoretic design and correctness verification, and statistical usage testing for quality certification. Cleanroom project development is based on incremental development and certification of the pipeline of user-function increments that accumulate into the final product.[42] Since the early pilot projects in 1987 and 1988, more than a dozen projects have been completed with a total size of more than half a million lines of code. The average defect rate found in first-time execution was 2.9 defects per thousand lines of code (KLOC), which is significantly better than the industry average.[42] To facilitate the adoption of Cleanroom, a phased implementation approach has recently been proposed.[43]

Design paradigms. The object-oriented approach to design and programming, which was introduced in the 1980s, represented a major paradigm shift in software development. This approach will continue to have a major effect in producing software for many years to come. Like the paradigm of structural design and functional decomposition, the object-oriented approach will become a major cornerstone in software engineering. Its influence on software reuse and productivity has already proven to be profound. To date, a number of object-oriented development projects have been successfully completed in the industry (for example, see Basili et al.[23]). Several of the finished products are integrated support environments for object-oriented development. After the initial learning curve, these projects showed significant increases in productivity and quality.

Programming languages. Good programming practices and programming languages that support the object-oriented paradigm continue to evolve. Object-oriented languages such as Ada**, Objective-C**, C++, and Smalltalk usually have strong typing and can enable object coherence checks. The object-based language Modula-2 has similar characteristics in terms of pre- and post-condition checking. These characteristics will lead to better development quality. Our limited experience so far (for example, projects at the IBM Rochester development laboratory) has lent support to this argument.

Automated software synthesis is a term now used to describe the translation of very-high-level languages (VHLL) into machine code. Current research in this area could bring a breakthrough in software development in terms of both productivity and quality. In software development, the higher the level of the programming language used, the more true productivity achieved (the more functionality implemented by the same amount of source code). When the automated software synthesis technology is mature and transferred, very-high-level languages can be used as the development languages.

Development environments. The last decade has also seen a significant improvement in development environments and tools. Integrated support systems have enabled configuration management, complex library systems, and change control for large-scale development. For large developers most of these support systems are developed internally, but support-system products are available in the industry for different software platforms.

With the advent of powerful workstations, software development is shifting to distributed environments. With ample processing power on the individual developer's desk and the availability of CASE tools, the outcome is more efficiency, higher productivity, and better quality.

Software reuse. The software industry had been continuously "reinventing the wheel" in terms of code development. During the past several years, code reuse has been receiving the attention for which it was long overdue. Some developers have been systematically stepping up their efforts in reuse. By using proven design and code in new software projects, both productivity and quality can be increased. Reusable parts, however, must meet pre-established criteria and must be of excellent quality. Parts with latent defects, and which are reused widely, can have a disastrous effect on new products. Software reusable parts should be certified.

The object-oriented approach should make reuse easier. Low-level object-oriented reusable com-

ponents are available in the industry in the form of class libraries.

In software development, reuse should not be limited to code. Project experience, defect models, process models, and so forth, should be reused to the extent possible for effective development. In this broader sense of reuse, the Experience Factory approach established by the NASA Software Engineering Laboratory (SEL) can provide a good solution. As discussed earlier, the Experience Factory is a logical or physical organization that undertakes systematic learning and packaging of reusable experiences. It supports project development by acting as a repository for experience, analyzing and synthesizing the experience, and supplying the experience to various projects on demand. It evaluates experience and builds models and measures of software processes, products, and other forms of knowledge. It uses people, documents, and automated support to do so.[20,23] Other than use by NASA SEL, the Experience Factory approach is being experimented with and used by some developers. For instance, the software development laboratory of IBM Toronto for the past three years has established an experience warehouse, an early form of the Experience Factory, to facilitate the reuse of software experiences.

Human side of software quality

In the TQM philosophy, the human side of quality includes factors such as total participation, management commitment and leadership, employee buy-in and empowerment, and other social and cultural factors. Of these factors, perhaps commitment, both from bottom-up and top-down, is the most important. Without commitment from the entire organization, the chance for success is slim. Commitment can best be measured by individual behavioral changes. Townsend and Gebhardt[44] developed self-evaluation criteria for an organization to assess whether it is truly practicing TQM. The first question is on the behavioral changes of management. Specifically, if the decisions made to improve quality expect behavioral changes of the employees but not of the executives and management, the organization is practicing quality by proclamation instead of TQM.

In software development at the operational level, TQM can be viewed as the integration of project, process, and quality management. During the past few decades, management teams of successful software developers have accumulated vast experience in project and schedule management. To avoid cost and schedule overruns, schedule progress is often managed at the microscopic level. In contrast, there is much less experience in process and quality management, especially during the development cycle. Product and development managers must manage in-process quality in the way in which they manage schedule for quality improvement to happen. Quality, process, and schedule management must be totally integrated for software development to be effective. Some developers in the industry have started doing so. It may take some time, however, for this integrated software project management style to be ingrained in practice.

In software development, reuse should not be limited to code.

There are also social, psychological, and cognitive factors in software engineering that are related to quality improvement. Most software projects are group activities, involving all the complexities of group dynamics, communication networks, and organizational politics. The study of group behavior in software development is in its infancy, but it promises to improve our understanding of the development process, particularly at the front end (for instance, in requirements and design). From theory and experience, front-end improvement is vital to software quality and productivity.

For instance, requirements errors in software projects are well recognized as a deficiency area. One reason is because customers' requirements evolve over time. However, poor communication accounts for much of the problem. Research has shown that successful software development is a joint process in which the developer learns the application domain and user operations and the

user learns the design realities and available choices.[45]

Communication plays a significant role not only between users and developers, but also among members of the development teams. Improved communications will lead to reduction of error injection in various phases in the development cycle. Our experience indicated that as high as 20 percent of software defects are related to interface problems; causal analysis of these defects pinpoints communications, or the the lack of it, as the root cause. As a second example, it is not infrequent that inadequate documentation is blamed for project problems, whereas the real culprit is poor communication.[45] Many developers do not consider it possible to maintain internal documentation that is sufficiently current to meet their needs. They obtain their information through informal networks. This fact suggests that encouraging, cultivating, maintaining, and supporting such networks may be more effective for problem-solving.

A myth exists in software development that some super developers' productivity and performance could be as high as 20 to 1, compared to an "average" developer. Such super developers, however, are of a rare breed and perhaps can only be found in a ratio of one in two to three hundred (for example, see Bernstein[46]). Nonetheless, individuals' differences have long been recognized by development managers as a significant factor affecting productivity and quality of the product. Research[45] indicated that application domain knowledge is a principal factor in the wide performance differences among individual developers. This finding tends to refute the frequently held concept that software development is a domain-independent activity that can be abstracted and taught totally by itself. It argues for a certain degree of specialization among programmers and that education must be provided in terms of domain-specific knowledge.

A study of expert debuggers showed that the stereotype of these people as isolated software "freaks" is not true.[47] The best debuggers have excellent communication, negotiation, team-building, and other social skills. They generally have a clear vision of the purpose and architecture of the system. They typically cultivate an extensive network of experts upon whom they can call.

It is apparent that the sociology of software and cognitive psychological studies in the context of software development indicate the strong possibility of more effectiveness. This research area is still in its infancy. The challenge to the software industry is how to systematically develop knowledge in this area, package the knowledge and findings, and then transfer the knowledge to development organizations and put it into practice. The task is not easy, but the benefits could be invaluable.

What is measured is improved.

Data, measurement, and model

What is measured is improved. Data and measurements are the most basic prerequisites for the improvement and maturity of any scientific or engineering discipline. Yet, in the discipline of software engineering, this area is perhaps one that has many critical problems and one that needs concerted effort for improvement.

The use of measurements, metrics, and models in software development assumes the availability of good data. In fact, the poor quality of data is a large obstacle in quality improvement. In general, data gathered during the formal machine testing phases are more accurate than those collected at the front end, such as requirements analysis and design reviews. Some companies do not even collect data at the front end of the development process, and for them in-process quality management means only monitoring data during the formal testing phases. To enhance data accuracy, a good tracking system for the entire development process must be in place, and the system must address the data validation issue. Such a system enables data-based decision-making for project and quality management. The amount of data to

be collected and the number of metrics to be used need not be overwhelming. It is more important that the information extracted from the data be focused, accurate, and useful.

Measurements for software projects, therefore, should be well thought through before being put in use. Metrics that are arbitrarily established could be harmful to the quality improvement effort of a company, and there are numerous examples of this sort in the industry. Each metric used should be subjected to the examination of basic principles of measurement theory; for example, the measurement scale, the operational definition, and validity and reliability issues should be well thought out. Validity refers to the ability of the metric to measure the parameter it is intended to measure, and reliability refers to the consistency of the measurement in operation. The draft of the IEEE standard for a software quality metrics methodology[48] even includes factors in addition to validity and reliability (for example, correlation, tracking, consistency, predictability, and discriminative power).

Measurement is associated with modeling. We must base measures on models to determine if the process or project is performing as planned. If multiple metrics are used, they must be in support of or an integral part of the model(s) used. Only an integrated approach, in the context of models, can effectively translate the piecemeal information from each metric to a coherent body of knowledge about the quality of the product or the progress of the project. In the past, measurement has been metric-oriented, rather than model-oriented. In other words, it has involved collecting data without an explicit goal, model, and context.

One such integrated approach is the Goal/Question/Metric (GQM) paradigm.[21,49] GQM is a mechanism for defining and evaluating a set of operational goals, using measurement. It represents a systematic approach for tailoring and integrating goals with models of the software processes, products, and quality perspectives of interest, based on the specific needs of the project, the customer, and the organization. The goals are defined in an operational, tractable way by refining them into a set of quantifiable questions that are used to extract the appropriate information from the models. The questions and models, in turn, define a specific set of metrics and data for collection and provide a framework for interpretation.

For example, when analyzing the effectiveness of the front-end defect removal of a project, development managers may collect defect data from design reviews and code inspections and obtain estimates of the size of the project. Defect rates can then be calculated. But they may be unable to interpret the data in a meaningful way because high defect rates could indicate effective reviews and inspections or high error injections, and low defect rates may indicate poor reviews and inspections or low error injection. However, if inspection process quality data are also collected and the two metrics (defect rate and process conformance) are looked at together within the context of a defect model, we can extract useful information. For instance, the best-case scenario occurs if inspection process quality increases and defect rate decreases, when compared with the model or with the previous release of the same product. In contrast, if inspection process quality decreases and defect rate increases, that would be the worst-case scenario, indicating the error injection may be higher and yet not enough rigor was put into the review process.

As another example, most product managers collect weekly defect arrival data during the formal machine testing phase. However, to interpret the data properly and to determine if the quality of the current project is really improving, a model with sufficient contextual information needs to be in place. The model should include information such as the size and defect measures, the normalized (to size) desirable model curve that is empirically validated by previous products, the actual defect arrival pattern of the predecessor product, assumptions and information about test effectiveness, and so forth.

We have begun to see more organized approaches to measurement—approaches based on models and driven by goals. For example, Motorola, Inc.[50] and Hewlett-Packard Co.[7] use GQM as the basis for their measurement approaches. Other mechanisms for defining measurable goals are also in use, for example, the software quality metrics approach.[51] The application of tools and techniques from traditional quality engineering in software development has also emerged (for example, see Kan[52]). We believe that when the organized approach to measurement and model-

ing is widely used, the resultant quality impact will be highly significant.

With regard to models in software engineering, cost and schedule models have moved from research and development into application. Models on software quality have also emerged. They can be broadly classified into three categories, each for a separate purpose: the reliability models, for reliability assessment and projection; the quality management models, for managing quality during the development process; and the complexity models and metrics, which can be used by software engineers for quality improvement in their work.[53]

Software reliability modeling is more mature than the other two types, and numerous software reliability models exist. Simply put, reliability models treat the software product as a black box, monitor its external behavior, and use sophisticated statistical extrapolation methods to predict its reliability.

Quality management models, which are still in their development and maturing phase, emerged from the practical needs of large-scale development projects. Examples include the Rayleigh model, the Remus-Zilles model, the phase-based defect removal model, and several tracking models during the testing phases.[53-55]

In contrast to reliability models, the complexity models tend to explain quality (or defects) from the internal structure and complexity of the software. Examples of complexity models and metrics include the lines of code count metric,[56] Halstead's software science,[57] McCabe's cyclomatic complexity,[58] and several structural metrics and models. In addition to these module complexity models, recently structural models have also been developed and used. For instance, the system complexity model developed by Card and Glass,[59] which defines system complexity as an additive sum of structure complexity (intermodule) and data complexity (intramodule), is quite promising.

These models, when properly selected and used, can yield tremendous benefits in software development quality. When the discipline of software engineering becomes more mature, we expect to see increased use of measurements and models, with increased benefits as a result.

Conclusion

We have discussed the definition of software quality, the total quality management (TQM) approach to quality improvement, and the advancements on many fronts of software engineering as they relate to the key TQM elements: customer focus, process, the human side of quality, and data, measurements, and analysis. In this modern quality era, TQM is widely embraced by numerous industries. However, the gap between TQM as a management style and the operational quality improvements in specific engineering disciplines is seldom understood and often ignored. We have tried to show the relevance of TQM in software development by discussing software-specific topics and their progress in the TQM framework.

Process, product, and quality models and other forms of structured experience have been defined, used, or accumulated, and will continue to aid in the practical engineering of software. New technologies (object-oriented paradigm, automated software synthesis, etc.) have emerged. Existing technologies are becoming better focused (customer feedback process, prototyping, DPP), more disciplined (reuse, design reviews, measurement approach), and more practical (formal development methods). The new advancements in social psychology and software sociology will enrich software engineering. However, significant challenges remain. Transferring software technologies into development organizations and bridging the gap between the state of the art and the state of practice needs concerted efforts by both researchers and practitioners. The use of quantitative approaches in software development, with feedback and learning through model-based measurements, needs to become ingrained in practice.

To bring about TQM in software development, we must take a systematic engineering approach to address the improvements of the numerous elements of software engineering, many of which we briefly covered in this tutorial. In contrast, the holistic TQM framework will aid software engineering to mature. Software engineering must move in this direction to become a true engineering discipline and to meet the increasing demands from society for effective development with high-level quality.

Finally, like software development, software quality is a domain-specific expertise. Software management and software developers are responsible for the implementation of software quality techniques and quality process. Software quality engineers play a very important role in process improvement, measurement, analysis, evaluation, and recommendation. Upper management leads the quality improvement direction and ensures a constant focus on continuous improvement. Significant improvement in software quality can be realized and sustained only through the total participation of all who are involved.

Acknowledgments

The authors wish to thank A. J. Montenegro for the invitation to write this tutorial and the referees for their comments.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of the U.S. Department of Defense or Stepstone, Inc.

Cited references

1. V. R. Basili and J. D. Musa, "The Future Engineering of Software: A Management Perspective," in *Computer* **24**, No. 9, 90–96 (September 1991).
2. F. P. Brooks, *The Mythical Man-Month*, Addison-Wesley Publishing Co., Reading, MA (1975).
3. B. Littlewood and L. Strigini, "The Risks of Software," *Scientific American* (November 1992), pp. 62–75.
4. J. Guaspari, *I Know It When I See It: A Modern Fable About Quality*, American Management Association, New York (1985).
5. P. B. Crosby, *Quality Is Free: The Art of Making Quality Certain*, McGraw-Hill Book Co., New York (1979).
6. J. M. Juran and F. M. Gryna, Jr., *Quality Planning and Analysis: From Product Development Through Use*, McGraw-Hill Book Co., New York (1970).
7. R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-wide Program*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).
8. C. Jones, "Critical Problems in Software Measurement," Version 1.0, Software Productivity Research (SPR), Inc., Burlington, MA (August 1992).
9. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Co., Reading, MA (1989).
10. W. E. Deming, *Out of the Crisis*, Center for Advanced Study, Massachusetts Institute of Technology, Cambridge, MA (1986).
11. A. V. Feigenbaum, *Total Quality Control: Engineering and Management*, McGraw-Hill Book Co., New York (1961).
12. A. V. Feigenbaum, *Total Quality Control*, McGraw-Hill Book Co., New York (1991).
13. K. Ishikawa, *What Is Total Quality Control? The Japanese Way*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985).
14. D. Shores, "TQC: Science, Not Witchcraft," *Quality Progress*, 42–45 (April 1989).
15. B. Zimmer, "Software Quality and Productivity at Hewlett-Packard," *Proceedings of the IEEE Computer Software and Applications Conference* (1989), pp. 628–632.
16. W. B. Smith, "Six Sigma: TQC, American Style," presented at the National Technological University television series (October 31, 1989).
17. M. J. Harry and J. R. Lawson, *Six Sigma Producibility Analysis and Process Characterization*, Addison-Wesley Publishing Co., Reading, MA (1992).
18. W. A. Shewhart, *Economic Control of Quality of Manufactured Product*, D. Van Nostrand Company, Inc., New York (1931).
19. V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," *Proceedings of the First Pan Pacific Computer Conference*, Melbourne, Australia (September 1985), pp. 379–398; also available as Technical Report TR-1519, Department of Computer Science, University of Maryland, College Park, MD (July 1985).
20. V. R. Basili, "Software Development: A Paradigm for the Future," *Proceedings 13th International Computer Software and Applications Conference (COMPSAC)*, Keynote Address, Orlando, FL (September 1989), pp. 471–485.
21. V. R. Basili and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, CA (March 30–April 2, 1987), pp. 345–357.
22. V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering* **SE-14**, No. 6, 758–773 (June 1988).
23. V. R. Basili, G. Caldiera, F. McGarry, R. Pajersky, G. Page, and S. Waligora, "The Software Engineering Laboratory: An Operational Software Experience Factory," *International Conference on Software Engineering* (May 1992), pp. 370–381.
24. R. A. Radice, J. T. Harding, P. E. Munnis, and R. W. Phillips, "A Programming Process Study," *IBM Systems Journal* **24**, No. 2, 91–101 (1985).
25. J. P. Womack, D. T. Jones, and D. Roos, *The Machine That Changed the World: Based on the Massachusetts Institute of Technology 5-Million-Dollar 5-Year Study of the Future of the Automobile*, Rawson Associates, New York (1990).
26. R. Likert, *The Human Organization: Its Management and Value*, McGraw-Hill Book Co., New York (1967).
27. R. Rudisill, "QCASE: A New Paradigm for Computer Aided Software Engineering," *Proceedings of the International Software Quality Exchange, 1992*, Juran Institute, Inc., Wilton, CT (1992), pp. 4A-19–4A-34.
28. J. R. Hauser and D. Clausing, "The House of Quality," *Harvard Business Review* **66**, No. 3, 63–73 (May–June 1988).
29. H. Gomaa and D. Scott, "Prototyping as a Tool in the Specification of User Requirements," *Proceedings 5th IEEE International Conference of Software Engineering* (March 1981), pp. 333–342.
30. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer* **21**, No. 5, 61–72 (May 1988).
31. V. R. Basili and A. J. Turner, "Iterative Enhancement: A

Practical Technique for Software Development," *IEEE Transactions on Software Engineering* **SE-1**, No. 4, 390–396 (December 1975).
32. P. H. Luckey, R. M. Pittman, and A. Q. LeVan, *Iterative Development Process with Proposed Applications*, Technical Report, IBM Corporation, Owego, NY (1992).
33. P. Jenkins, IBM Boca Raton, personal communications (1992).
34. C. L. Jones, "A Process-Integrated Approach to Defect Prevention," *IBM Systems Journal* **24**, No. 2, 150–167 (1985).
35. R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski, "Experiences with Defect Prevention," *IBM Systems Journal* **29**, No. 1, 4–32 (1990).
36. J. C. Knight and E. A. Myers, "Phased Inspections and Their Implementation," *ACM SIGSOFT Software Engineering Notes* **16**, No. 3, 29–35 (July 1991).
37. D. J. Smith and K. B. Wood, *Engineering Quality Software: A Review of Current Practices, Standards and Guidelines Including New Methods and Development Tools*, Second Edition, Elsevier Applied Science, New York (1989).
38. J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer* **23**, No. 9, 8–24 (September 1990).
39. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software* **4**, No. 5, 19–25 (September 1987).
40. R. W. Selby, Jr., V. R. Basili, and T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering* **13**, No. 9, 1027–1037 (September 1987).
41. M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc., New York (1992).
42. R. C. Linger, *Cleanroom Software Engineering for Zero-Defect Software*, IBM Cleanroom Software Technology Center Technical Paper, IBM Corporation, Gaithersburg, MD (1992).
43. P. A. Hausler, R. C. Linger, and C. J. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems Journal* **33**, No. 1, 89–109 (1994, this issue).
44. P. L. Townsend and J. E. Gebhardt, *Commit to Quality*, John Wiley & Sons, Inc., New York (1990).
45. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM* **31**, No. 11, 1268–1287 (1989).
46. L. Bernstein, "Notes on Software Quality Management," presented at the *International Software Quality Exchange*, San Francisco (sponsored by the Juran Institute), (March 10–11, 1992). Mr. Bernstein, Vice President of Operations Systems at AT&T, commented specifically on the ratio of super developers from his experience.
47. T. R. Riedl, "Application of a Knowledge Elicitation Method to Software Debugging Expertise," presented at the *Fifth Conference of Software Engineering Education*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (October 1991).
48. N. F. Schneidewind, "Report on the IEEE Standard for a Software Quality Metrics Methodology (Draft) P1061, with Discussion of Metrics Validation," *Proceedings of the IEEE Fourth Software Engineering Standards Application Workshop* (1991), pp. 155–157.
49. V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering* **SE-10**, No. 6, 728–738 (November 1984).
50. M. K. Daskalantonakis, "A Practical View of Software Measurement and Implementation Experiences within Motorola," *IEEE Transactions on Software Engineering* **SE-18**, No. 11, 998–1010 (1992).
51. B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," *Proceedings Second International Conference of Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA (1976), pp. 592–605.
52. S. H. Kan, "Applying the Seven Basic Quality Tools in Software Development," *Proceedings of the International Software Quality Exchange*, Juran Institute, Inc., Wilton, CT (1992), pp. 4A-35–4A-52.
53. S. H. Kan, "Software Quality Engineering Models," *Encyclopedia of Computer Science and Technology*, A. Kent and J. G. Williams, Editors (forthcoming, Marcel Dekker, Inc., 1994).
54. S. H. Kan, "Modeling and Software Development Quality," *IBM Systems Journal* **30**, No. 3, 351–362 (1991).
55. H. Remus and S. Zilles, "Prediction and Management of Program Quality," *Proceedings of the Fourth International Conference on Software Engineering*, Munich (1979), pp. 341–350.
56. C. Withrow, "Error Density and Size in Ada Software," *IEEE Software* **7**, No. 1, 26–30 (January 1990).
57. M. H. Halstead, *Elements of Software Science*, Elsevier North Holland, New York (1977).
58. T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2**, No. 4, 308–320 (1976).
59. D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).

Accepted for publication August 12, 1993.

Stephen H. Kan *IBM AS/400 Division, Highway 52 and NW 37th Street, Rochester, Minnesota 55901.* Dr. Kan is an advisory programmer in IBM Rochester's Development Quality and Process Technology department. He holds B.S. degrees in sociology and computer science, M.S. degrees in statistics and sociology, and a Ph.D. degree in demography from Utah State University. He joined IBM in 1987 and prior to that had been working as a computer programmer, statistician, and research scientist for eight years in academia and industry. He is a Certified Quality Engineer by the American Society for Quality Control. In his current assignment, his focuses are software quality strategy, software quality plans, supplier quality requirements, defect removal models and in-process quality, and software statistical analysis.

Victor R. Basili *Department of Computer Science, A. V. Williams Building, University of Maryland, College Park, Maryland 20742.* Dr. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science department at the University of Maryland, where he served as chairman for six years. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation, established in 1976. He

serves on the editorial board of the *Journal of Systems and Software* and is an IEEE Fellow. He has been editor-in-chief of the *IEEE Transactions on Software Engineering*, general chairman of the 15th International Conference on Software Engineering, program chairman for several conferences, including the 6th International Conference on Software Engineering in Japan, a member of the Computing Research Board, and a Governing Board member of the IEEE Computer Society.

Larry N. Shapiro *IBM Software Solutions Division, 555 Bailey Ave., San Jose, California 95141.* Mr. Shapiro is a program manager in the IBM Santa Teresa laboratory's Market-Driven Quality Strategy department. He holds a B.S. degree in mathematics from the University of Southern California. He joined IBM in 1960 in Applied Science and has held numerous management and technical positions in marketing, development, planning, and site management. In his current assignment, his focus is on the Santa Teresa laboratory's total quality progress and continuous improvement programs.

Reprint Order No. G321-5530.