Technical Report ITL-99-4
June 1999

US Army Corps
of Engineers
Waterways Experiment
Station

An Introduction to Software Quality

by Buvaneswari K. Venkataraman, William A. Ward, Jr.,
University of South Alabama

WES

Prepared for Headquarters, U.S. Army Corps of Engineers

Technical Report ITL-99-4
June 1999

An Introduction to Software Quality

by Buvaneswari K. Venkataraman, William A. Ward, Jr.

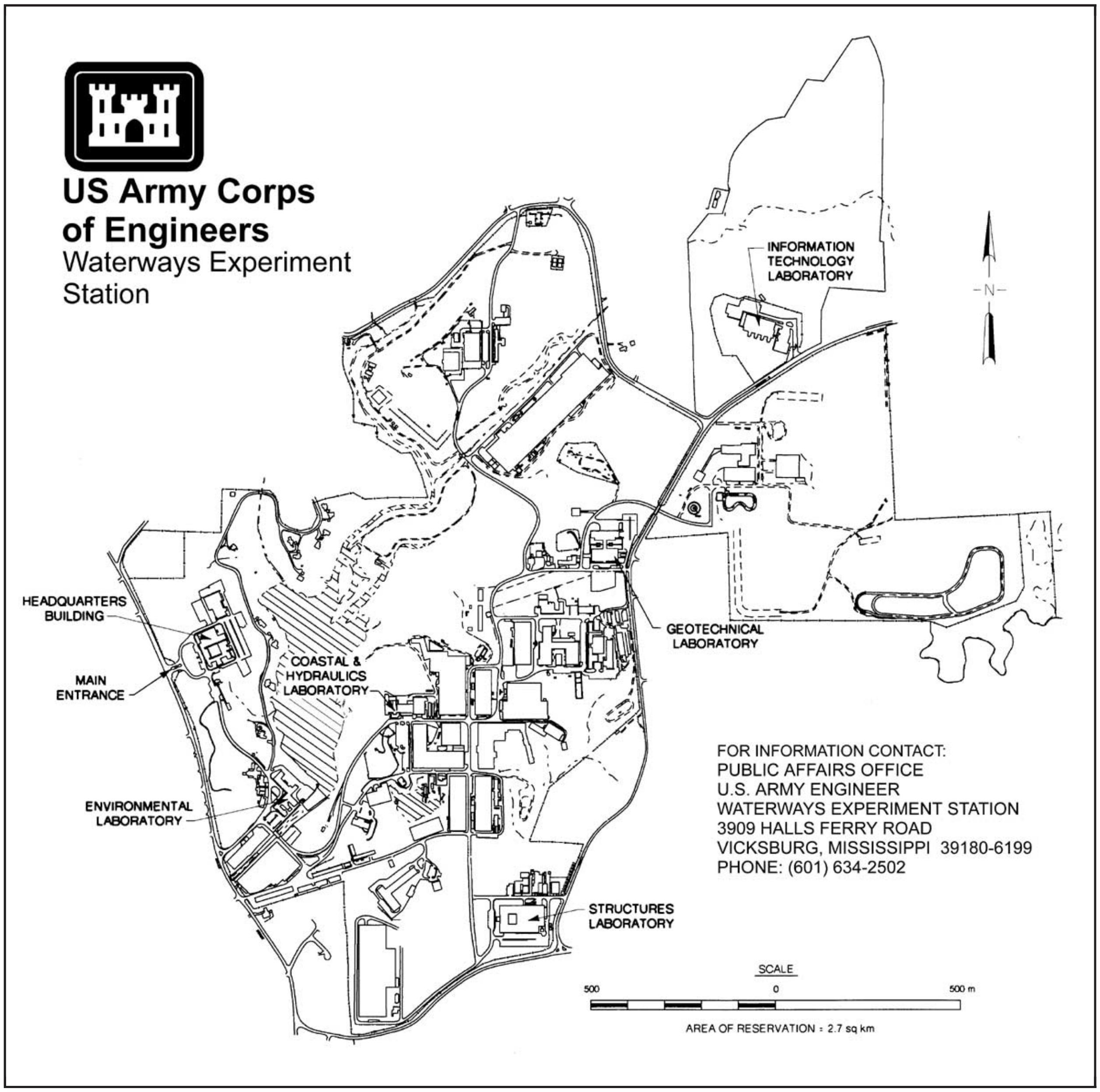
Faculty Court West 20
School of Computer and Information Sciences
University of South Alabama
Mobile, AL 36688

Final report

Prepared for U.S. Army Corps of Engineers
Washington, DC 20314-1000

Under Contract No. DACA39-93-K-0016

Monitored by U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, MS 39180-6199


US Army Corps of Engineers
Waterways Experiment Station
INFORMATION TECHNOLOGY LABORATORY
N
HEADQUARTERS BUILDING
MAIN ENTRANCE
COASTAL & HYDRAULICS LABORATORY
GEOTECHNICAL LABORATORY
ENVIRONMENTAL LABORATORY
STRUCTURES LABORATORY
SCALE
500
0
500 m
AREA OF RESERVATION = 2.7 sq km

FOR INFORMATION CONTACT:
PUBLIC AFFAIRS OFFICE
U.S. ARMY ENGINEER
WATERWAYS EXPERIMENT STATION
3909 HALLS FERRY ROAD
VICKSBURG, MISSISSIPPI 39180-6199
PHONE: (601) 634-2502

Waterways Experiment Station Cataloging-in-Publication Data

Venkataraman, Buvaneswari K.
An introduction to software quality / by Buvaneswari K. Venkataraman, William A. Ward, Jr. ; prepared for U.S. Army Corps of Engineers ; monitored by U.S. Army Engineer Waterways Experiment Station.
40 p. : ill. ; 28 cm. — (Technical report ; ITL-99-4)
Includes bibliographical references.
1. Computer software — Quality control. 2. Total quality management. I. Ward, William A. II. United States. Army. Corps of Engineers. III. U.S. Army Engineer Waterways Experiment Station.
IV. Information Technology Laboratory (U.S. Army Engineer Waterways Experiment Station) V. Title.
VI. Series: Technical report (U.S. Army Engineer Waterways Experiment Station) ; ITL-99-4.
TA7 W34 no.ITL-99-4

Contents

Preface

The production of this report was sponsored by Headquarters, U.S. Army Corps of Engineers, and monitored by the U.S. Army Engineer Waterways Experiment Station (WES) Information Technology Laboratory (ITL) under Contract No. DACA39-93-K-0016 from 1 January 1995 through 30 June 1995. The contract was monitored by Dr. Windell F. Ingram, Chief, Computer Science Division, ITL. Dr. N. Radhakrishnan was Director, ITL.

This report was prepared by Ms. Buvaneswari K. Venkataraman and Dr. William A. Ward, Jr., of the University of South Alabama.

At the time of publication of this report, COL Robin R. Cababa, EN, was Acting Directer of WES.

1 Introduction

Why Software Quality?

Software has become pervasive in modern civilization. Virtually all businesses use computers to perform billing, payment, inventory control, purchasing, process control, business forecasting, and numerous other crucial functions. Personal computers are in a large percentage of American households and are used to manage household expenses, file income taxes, and access Internet services, as well as for recreational purposes. These visible computer systems, however, are outnumbered by a host of invisible systems embedded in everything from appliances to automobiles. Poor performance or failure of these systems may result in annoyance to the customer, loss of customers, or even loss of life. The quality of the software controlling these systems is, therefore, an important concern. Some examples of the consequences of poor software quality serve to emphasize this point.

In the late 1970s, the U. S. General Accounting Office performed a study (Comptroller General of the United States 1979) of nine software development projects performed for the Department of Defense costing $6.8 million. That amount was distributed as follows:

a. $3.2 million for software delivered to the Government but never successfully used

b. $1.95 million for software never delivered

c. $1.3 million for poor quality software requiring extensive modification or ultimate abandonment

d. $198,000 for software requiring modification before being used

e. $119,000 for software used as delivered

This study has been criticized as being unrepresentative or too small in scope, but the problems it noted are still with us.

Space programs provide spectacular examples of software failure. Less than five minutes after its launch on July 22, 1962, the rocket carrying the U.S. Mariner I Venus probe had to be destroyed because of an error in the programming of a ground guidance control computer (Sethi 1996). On June 4, 1996, because of a software exception in an on-board computer, the French rocket Ariane 5 exploded less than one minute after launch (SIAM News 1996). The exception was caused because the author of the program believed that the particular program variable involved would never be subject to an overflow error, and so did not include code to trap and handle that condition. Ironically, the program involved was written in Ada, a language which claims program reliability and robustness as strengths.

On a more somber note, several individuals were fatally injured due to a malfunction in the Therac 25, a medical device for administering therapeutic radiation to patients (Joyce 1987). The machine was designed to operate in two modes, brief high radiation dosages and longer low dosages; the original, non-computerized versions of these machines had mechanical interlocks to prevent a long high radiation mode. As a cost saving measure, the Therac 25 implemented this interlock in software; under certain combinations of inputs, the software interlock could (and did) fail. Other examples of software disasters are noted by Brown, Earl, and McDermid (1992) and by Gibbs (1994).

Definition of Software Quality

The International Standards Organization (ISO) formally defines quality as "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs." More generally, people think of quality as conformance and compliance to specifications continuously and consistently. According to Garvin (1984), quality is multifaceted and can be viewed from several perspectives, the transcendental view, the user-based view, the product-based view, the manufacturing view, and the value-based view. The transcendental view equates the quality of a product to "innate excellence" that cannot be described but can only be recognized if exposed to it. If this view is correct, then quality is inherently unmeasurable and the use of quantitative methods to improve quality is a waste of time. Engineers in general, and software engineers in particular, deny this view.

The user-based approach views the quality of a product as the ability of the product to provide maximum satisfaction to the user by being durable and performing to the best of its capability. The expectations of users (including businesses as well as individuals) have to be met for a product to be successful. The number of businesses that depend on purchased software has multiplied dramatically, and their requirements range from consumer products to space applications. Due to the critical nature of many of these applications, it is imperative that software consistently meets its specified functions. Thus, the pervasiveness of software over the past decade in virtually all walks of life has brought about an increased emphasis on the user-based view of software quality.

Also, the increased use of PCs and the multiplicity of software packages for PCs has increasingly tied the user-based view to the product view and the value-based view (to be discussed below).

The product-based view considers that quality is dependent on the inherent characteristics of a product and can be quantified based on the presence or absence of a number of attributes. Organizations all over the world allocate considerable resources for their quality assurance programs due to the belief that a product's market share and profitability are firmly tied to its quality. An organization's reputation and market gains are thus a measure of the quality of its products. Good quality also results in lower rework costs, thereby improving profits.

The manufacturing view focuses on “conformance to requirements” and defines quality as being a function of the process quality. Therefore, it is not surprising that in almost every organization involved in manufacturing, there are groups that exclusively monitor and control quality. Like any other manufacturer, a software development organization also has independent teams that thoroughly test/audit the software product, removing every possible error or bug before it reaches the customer (Brooks 1995). In fact, these teams serve as the customers for the product development team. Such teams are indispensable for organizations that make a conscious effort to deliver products of high quality. Quality is of strategic importance to organizations, and by constantly delivering products of the highest quality, they can retain their competitive edge in the market.

The value-based approach defines quality as that which provides the required performance at affordable and acceptable costs. Therefore the other aspects of software quality must be obtained without inordinate expenditure. As will be seen, this implies defect detection and repair early in the software life cycle, when these activities are cheaper to perform. This means that the quality of the development process contributes significantly to product quality, and so the value-based view is influenced by the manufacturing view.

The rest of this report is organized as follows. Chapter 2 discusses software quality improvement through better evaluation methods and models. Chapter 3 deals with measuring software quality. Improved management and development processes and standards are the topic of Chapter 4. Use of quality tools and environments to develop quality software is discussed in Chapter 5. Chapter 6 presents some concluding remarks.

2 Software Quality via Better Quality Evaluation

Almost every organization has its own internal standards that provide a guideline for measuring and monitoring quality. Standards increase the level of understanding of the process by the project members, thereby promoting better communication. In addition to standards, organizations need clearly defined quality models to effectively meet the demands from customers. Such models list major attributes that a high-quality software product should possess (e.g., reliability and maintainability). The model may then break down the major attributes into subattributes to produce a tree-type hierarchy. Another possibility involves relating the major attributes to the subattributes in such a way that one or more of the latter are each shared with several of the former to produce a graph (instead of a tree). The model may also be accompanied by an evaluation methodology indicating what should be measured to produce a score for each of the the attributes and ultimately for the the entire software product.

Models for software quality evaluation facilitate clearer understanding of the entire process of software engineering. Quality models are useful in predicting reliability, in managing quality during the development process, and in assessing the complexity of software (Kan, Basili, and Shapiro 1994). Generally, reliability models use statistical methods to measure the reliability of software. Most quality management models focus on defect-removal and defect-tracking during the development process. The complexity models look at the structure of the software to determine quality. Every software development organization has a quality evaluation model whether it realizes it or not. Often the model is unconsciously selected as a side effect of some other business decision (e.g., setting a deadline or allocating personnel to particular activities). It is important for such organizations to explicitly develop a quality model that best suits their interests and implement it. Over the past few years, many quality models have been built by a number of researchers to aid in this effort.

McCall's Model

One of the earliest quality models was proposed by McCall (McCall, Richards, and Walters 1977); this model describes quality as being made up of a hierarchical relationship between the quality factors, quality criteria, and quality metrics. McCall’s systematic approach to quantify quality is as follows.

a. Determine all of the factors that would have an effect on the software quality.

b. Identify the criteria for judging each factor.

c. Define metrics for each of the criteria and establish a normalization function that defines the relationship between the metrics of all the criteria pertaining to each factor.

d. Evaluate the metrics.

e. Correlate the metrics to a set of guidelines that every software development team could follow.

f. Develop recommendations for the collection of metrics.

The term “quality factor” defines some key characteristic that a product would exhibit. “Quality criterion” represents some attribute of the quality factor that defines the product. “Quality metric” denotes a measure that can be used to quantify the criterion. McCall identified a number of criteria like traceability, simplicity, machine-independence, storage efficiency, operability, error tolerance, expandability, conciseness, etc. that could be associated with the quality factors. The metrics he developed involved questions dealing with the degree of compliance to the criteria and had either a “yes” or a “no” for an answer. The responses for these questions would be highly subjective and generally difficult to interpret into reasonable indicators of quality. McCall's quality factors and their associated critera are shown in Table 1; note that some criteria are shared by more than one factor.

ISO 9126

Recently, a new standard for software product evaluation, ISO 9126, has been developed by the ISO (1992). This standard has identified six basic quality characteristics that must be present in a quality software product (Kitchenham and Pfleeger 1996). The standard also provides a sample decomposition of these basic characteristics into subcharacteristics; these are listed in Table 2. An alternate decomposition of the ISO 9126 basic characteristics is described in Tervonen (1996). Tervonen's model combines aspects of the software quality metrics (SQM) model (McCall, Richards, and Walters 1977) with ISO 9126 to

Table 1
McCall's Software Quality Factors and Their Associated Criteria[1]

Factor	Criteria
Correctness	Traceability, completeness, consistency
Reliability	Consistency, accuracy, error tolerance
Efficiency	Execution efficiency, storage efficiency
Integrity	Access control, access audit
Usability	Operability, training, communicativeness
Maintainability	Simplicity, conciseness
Testability	Simplicity, instrumentation, self-descriptiveness, modularity
Flexibility	Self-descriptiveness, expandability, generality, modularity
Portability	Self-descriptiveness, software-system independence, machine independence
Reusability	Self-descriptiveness, generality, modularity, software-system independence, machine independence
Interoperability	Modularity, communications commonality, data commonality

[1] From McCall, Richards, and Walters (1977)

Table 2
ISO 9126 Software Quality Characteristics and Subcharacteristics[1]

Characteristic	Subcharacteristics
Functionally	Suitability, accuracy, interoperability, security
Reliability	Maturity, fault tolerance,recoverability
Usability	Understandability, learnability, operability
Efficiency	Time behavior, resource behavior
Maintainability	Analyzability, changeability, stability, testability
Portability	Adaptability, installability, conformance, replaceability

[1] Adapted from a figure in Kitchenham and Pfleeger (1996)

produce the SQM synthesis model. Unlike the previous two models, this model has three levels; the ISO characteristics (a) are decomposed into criteria (b) which are in turn broken down into factors (c). Tervonen proposes associating each of the factors with several checklists. These checklists are made up of questions that a knowledgeable software inspector may answer regarding features in the code. The scores for these checklists would ultimately be combined to produce quality measures for the factors, the criteria, and finally the characteristics. Tervonen also discusses the use of software tools to support the assessment process.

Other Models

A model based on a variation of the value-based view of quality has been proposed by Simmons (1996). In her study, nine Australian organizations were surveyed to determine how they measured the effectiveness of information technology projects. The aspect of software quality addressed here was not the quality of code itself, but what the software did to improve business. The results indicated that six of the nine believed that measuring this effectiveness was important. All nine generally wanted a quantifiable financial metric, and some even used alternative nonfinancial measures. Based on these results, the author constructed a framework for categorizing benefits according to whether they increased efficiency, increased effectiveness, added value, produced a marketable product, or provided necessary infrastructure for other activities. Simmons noted that economic measures were generally used for the first, second, and fourth categories, but that there were no direct measures for the third and fifth.

A general framework for constructing software quality models has been proposed by Dromey (1996) who notes, “We cannot build high-level quality attributes like reliability or maintainability into software. What we can do is identify and build in a consistent, harmonious, and complete set of product properties (such as modules without side effects).” The problem then becomes one of linking the measurable product properties to the high level attributes. To solve this problem, he suggests the use of four categories of quality-carrying properties: correctness properties, internal properties, contextual properties, and descriptive properties. A five-step process for constructing models is based on these properties.

a. Specify the high-level quality attributes (e.g., reliability or maintainability).

b. Determine the various components of the product at an appropriate level of detail (package, subroutine, statement).

c. For each component, determine and categorize its most important quality-carrying properties. For example, a subroutine component should have the “side-effect free” quality-carrying property; this property is classified as “contextual.”

d. Propose links relating the quality-carrying properties to the quality attributes, or, alternatively, use links from the four property categories to the attributes.

e. Iterate over the above steps, using a process of evaluation and refinement.

Dromey illustrates the use of this procedure by constructing an implementation quality model, a requirements quality model, and a design quality model.

3 Software Quality via Better Measurement

There are a variety of decision support systems and information systems that are available off-the-shelf for predicting the cost, schedules, and other strategic details whenever a software project is initiated. However, quality itself cannot be predicted just by employing such tools. Therefore, it is essential that organizations clearly define and quantify the quality that is desired from the software product. To make realistic assessments about the quality of a software product, it is imperative that measurements be performed. This enables organizations to establish and regulate the levels of acceptable quality, predict quality, and to continuously strive to improve quality. Collection of data about the process and the product is the primary way of monitoring quality in a number of organizations. Generally, an organization's requirements and goals dictate the types of measurement that would be suitable.

Halstead (1977), in his pioneering work on software physics, proposed a number of methods for measuring software. He quantitatively evaluated key characteristics or metrics that facilitate effective measurement of a program. Some of these are program level, intelligence content, modularity, program volume, redundancy factor, branch count, total operators and operands, and program length. These metrics not only facilitate software measurements but also help in making realistic estimates of requirements for future programming projects. The other benefits of using such exhaustive metrics are initial error rate assessment, programming language evaluation, and the effects of writing modular code. In any programming project, a great deal of time and effort are spent on troubleshooting errors in the code. Hence, an understanding of the initial error rate would definitely help in scheduling the release date of a software product. Decisions about the program implementation language could be made with a greater degree of awareness with metrics like the program level and the program volume.

Boehm (1987) identified ten metrics in the software development process. Some of those that pertain to software quality are as follows:

a. The cost of correcting an error after delivery to the customer is 100 times more than for correction of an error that is detected earlier.

b. About 60 percent of the software development time is spent on requirements and testing. Only 25 percent of the time is spent on testing.

c. The most cost-effective way of detecting and correcting software error is inspection, which can capture about 60 percent of all errors.

As far as the size of the software is concerned, the most important metrics are the noncomment lines of code and function points (Fps). The number of lines of source code was the primary metric used to measure productivity until 1979 when Albrecht of IBM developed the FP metric (Jones 1997). FPs measure the size of a system in terms of its constituent components, namely, its inputs, outputs, inquiries, and files (Kemerer and Porter 1992). Depending on the functionality that needs to be implemented in a system, the number of source lines of code that have to be written could be found out. Reuse of parts of code has a dramatic effect on productivity as well as quality. To evaluate the quality of a software product, the most common metric employed is the number of defects per thousand lines of code (KLOC).

The other commonly employed metric is complexity (Arthur 1993). Complexity could be measured using McCabe's (1976) cyclomatic complexity (CC). This is a mathematical technique to identify the constituent modules of a software product that would be difficult to test and maintain. It is widely believed that a CC measure of 10 or less results in zero-defect software and would also facilitate reuse and maintenance. These factors, no doubt, influence the end quality of a software product. The actual steps involved in developing measurement procedures could be summarized as follows:

a. Establish the objectives of the measurement clearly.

b. Develop models for the process of collecting data.

c. Identify the resources (people, training, tools) needed to accomplish the measurement.

d. Create and implement the measurement process.

e. Evaluate the results of the measurement.

f. Constantly monitor the measurement process for continuous improvement.

During the software testing process, code inspections and walk-throughs are generally performed to evaluate the quality of software. The average lines of code inspected, the average inspection rate, the average effort per KLOC, and the defect-removal efficiency are some of the metrics that are typically collected during this stage (Barnard and Price 1994). Even if a number of measurements are performed and metrics data are collected, they are of absolutely no use unless rigorous analyses are performed on the data to determine ways to continuously improve the development process.

Obviously this chapter can only touch on the issues and concepts related to software measurement. Interested readers should refer to Pfleeger (1997) as a starting point for further study.

Choice of Metrics

It is not necessary to employ an overwhelming number of metrics. The quality control team has to select only those metrics that would yield relevant, accurate, and useful information about the process and the product. Metrics that are established arbitrarily could have a negative impact on the quality procedures of an organization. Each metric, before selection, has to be subjected to severe scrutiny with regard to measurement scale, validity, reliability, and predictability (Kan, Basili, and Shapiro 1994). A valid metric is that which has the capability to measure the intended parameter, and reliability refers to the ability of the metric to consistently measure the parameter correctly. Once all the metrics are collected, they have to be viewed in an integrated context to clearly discern the effect on the total quality of the product. This approach is feasible only if a coherent, integrated model of the process, product, and desired quality is built.

The Goal/Question/Metric paradigm (Basili and Weiss 1984) is a technique for establishing and evaluating a set of operational goals. It is based on the fact that the goals of an organization are the driving forces behind its quality standards. This is a methodical approach to integrate the goals or the objectives with the process and quality models, based on the specific requirements of the project, the customer, and the organization. Once the goals are defined, they are translated into a set of quantifiable questions that would extract information from the models (Kan, Basili, and Shapiro 1994). With these questions and the models, metrics are established that enable data collection and interpretation.

Metrics for the Software Life Cycle

Metrics collection should encompass each and every stage of the software development process. Quality consciousness must be instilled into the developers' minds in such a way that quality checks are performed at every stage of the software development life cycle. Most of the defects in a software product are introduced during the requirements and the design stage. The major difficulty that is faced by organizations after the completion of projects is that the systems do not conform to the user's specifications. Consequently, validation checks that would focus on specific aspects of the system at every stage of the software life cycle would be beneficial.

The requirements stage generally yields descriptions of the expected inputs and outputs of the system. Therefore, the parameters that would be of interest at this stage are correctness, consistency, and completeness. The next step, design, should be validated with the degree of modularity, coupling, and cohesion in the design document. At the end of the analysis phase, the efficiency, complexity, appropriateness, ease of understanding, and ease of implementation of the design are reviewed. Once the design has been implemented into code, thorough testing is performed to validate the parameters, conformance to standards, documentation, maintainability, and reliability. The testing phase of the software life cycle generally is subject to rigorous evaluation as to the feasibility of the test procedures, thoroughness, and functionality. During maintenance, operational evaluation is performed that considers availability, performance, and effectiveness.

Metrics for Object-Orientation

In recent years, the object-oriented paradigm has been regarded as yielding significant improvement in the quality of software. The basic characteristics of the object-oriented methodology, encapsulation, inheritance, and polymorphism are instrumental in achieving better quality. The analysis, design, and testing stages are inextricably woven together in this paradigm so testing is generally incremental. Inspection, code walk-throughs, and compilation are some of the techniques used in testing object-oriented software. Due to the encapsulated nature of the software, several levels of testing may be done. Class testing deals with developing test cases and thoroughly testing the basic unit of the software. In cluster testing, the focus is on the interaction among cooperating classes. System testing deals with testing after integration of all components. The major goal of object-orientation is developing reusable, extensible, and reconfigurable code (McGregor and Korson 1994). Therefore, any testing methodology for object-oriented software has to address this goal. Metrics have to be developed to evaluate the reusability, interactive behavior, structure, and functionality of the software.

Combining Individual Metrics

Assuming that an appropriate software model has been used, and that metrics have been gathered for each of the lowest level features in the model, the next step is to combine the scores at the lowest level to produce scores for the higher levels and, ultimately, an overall score. An appropriate technique to accomplish this is the multi-element component comparison and analysis (MECCA) methodology (Ulvila and Brown 1982). The application of MECCA requires a hierarchy of attributes, which should already be provided by the quality model. Every attribute is assigned a weight indicating its importance in the software quality evaluation; these are assigned so that the sum of the child weights sum to one for each individual parent. Scoring is accomplished by assigning scores to each of the leaf nodes; a parent's score is the weighted average of the scores of its children. This method has been successfully used by Magnavox Electronic

Systems Company (1990) in their evaluation of software development environments for Version 1 of the Advanced Field Artillery Tactical Data System.

4 Software Quality via Better Processes

The quality of the final product in every manufacturing discipline is directly dependent on the quality of the process. Standards are generally established that clearly explain the sequence of steps to be followed to create the product. Adherence to the steps by every member of the manufacturing team is an important criterion in the successful implementation of the standards. This is true in the case of software development as well. Establishment of a well-defined process infrastructure is instrumental in facilitating continuous improvement of the process.

The software process maturity (Paulk et al. 1993) of a software development organization is defined as "the extent to which a specific process is explicitly defined, managed, measured, controlled, and effective." The software process capability is defined as "describing the range of expected results that can be achieved by following a software process." Immature organizations generally have the following problems.

a. No organized standards base.

b. Existence of standards not known to many developers.

c. Procedures and standards exist but are not enforced rigorously.

d. Standards followed most of the time but schedule crunches cause slack in testing.

Crisis management is the norm in such organizations. Whenever there is a dispatch schedule that has to be met, it is done at the cost of testing and reviews. The quality of the final product would be compromised in such situations. In mature organizations, effective planning is performed to establish well-defined procedures for the software development process, and the procedures are religiously followed throughout.

The remainder of this chapter discusses software process quality from several increasingly specific perspectives. Total quality management (TQM) is a management approach to quality that may be applied by organizations in any field to produce quality products. While TQM is a rather general concept, the next section addresses the ISO 9000 Standard, which provides standardized guidance in the areas of production and management for achieving quality. The Capability Maturity Model (CMM) (Paulk et al. 1993), developed by the Software Engineering Institute (SEI), is the subject of the next section. Its guidance is specific to software development organizations and teams. Finally, the Personal Software Process (PSP) is discussed; it addresses the performance of individual software engineers.

Total Quality Management

TQM is another popular concept that has been adopted by many companies to achieve long-term success. Specific implementation methodologies for realizing TQM have been proposed in the past decade (Kan, Basili, and Shapiro 1994). W. E. Deming, one of the best known quality advocates in recent years, treats quality as a way of life. His philosophy stresses the importance of management commitment toward setting quality objectives. Some of Deming's principles are:

a. Quality is the responsibility of the management.

b. Quality should not be compromised at any expense.

c. Defects, if present, are caused by the system rather than the workers.

d. The process should be managed in such a way that quality is built into the system.

e. Incentives, rankings, and appraisals to judge individual achievement basically threaten and frighten the workers. This would adversely affect their feelings of security and motivation. Such merit systems have to be eliminated in order to foster cooperation among the workers.

f. Vendors and subcontractors should be chosen based on their ability to deliver quality products and not on the sole basis of cost.

Recently, several novel techniques have been deployed for total quality management during every stage of the software development life cycle (Haag, Raja, and Schkade 1996). Statistical process control is one such technique that could be widely used to gain insight into the software development process (Card 1994). Techniques such as this are dependent on measuring the quality of software during its development so that improvements can be made. The significance of quantitative evaluation of software is thus evident.

Quality management and compliance to quality standards are influential factors that have started having a substantial effect on the software industry all over the world. Due to the customers' increased awareness of these global standards, it has now become imperative that organizations that were indifferent to implementing good quality practices take a serious look at their software process.

ISO 9000 Standard

The other important factor that influences quality management principles in the software development process is the ISO 9000 standard developed by the ISO. More than 50 nations all over the world have adopted the ISO 9000 standards (Schmauch 1994). The main driving force behind the global acceptance of this standard lies in its adoption by Europe. This has put a lot of pressure on manufacturers worldwide who are interested in the European market. The development and deployment of international standards such as the ISO 9000 have brought about a universal measure of quality in many products. The basis for the ISO 9000 standard is the premise that a right production and management system produces the right product. These standards are applicable to any type of environment because they are generic models and not specific to any type of business. The ISO 9000 standard places much emphasis on documentation of each and every procedure that exists in the process. Good documentation ensures a greater degree of control, auditability, verification/ validation, and process improvement (Schmauch 1994).

ISO 9000 certification is granted to an organization when it demonstrates that its quality system conforms to the ISO standard during an audit by a third party accredited registrar. There are five sections to the ISO 9000 standard, and, based on the type of business they are in, organizations decide which standard to use. There are 20 standards elements specified by the ISO standard; they are described in the following paragraphs. Most of these standards elements are applicable to the software development environment directly. However, some of them require proper interpretation in order to be useful in that environment.

a. *Management responsibility*. A quality policy must exist for the entire organization and must be understood and implemented by every employee. Management must show its commitment to quality by authorizing a high-level manager who is responsible for the quality system and its periodic review. This is applicable to software development organizations as well.

b. *Quality system*. A clearly defined quality system should be present to ensure that a product meets its specified requirements. Generally, organizations develop a quality manual that documents their quality procedures.

c. *Contract review*. The organization should have methods to ensure that the customer requirements concerning the product are understood thoroughly and are agreed upon by both parties.

d. *Design control*. Well-documented procedures should exist for the design process and design review and for design change control.

e. *Document control*. All necessary documents should be available to the right persons at the right time. Documents have to be current, and obsolete documents have to be removed. The documents produced during the different stages of the development process have to be monitored, updated, and approved by the document control team.

f. *Purchasing*. In addition to maintaining quality in-house, if there are components that are purchased from outside vendors, there should be some system to evaluate the vendor based on previous history of performance, quality, and timeliness.

g. *Purchaser-supplied product*. If some components are supplied by the customer, there should be some means of ensuring safe storage and maintenance of those parts.

h. *Product identification and traceability*. At any stage during the development process, it should be possible to trace which component went into which final product. This element of the ISO standard is oriented toward a manufacturing environment. Nevertheless, it is applicable to the software process where component parts of a huge software product are invariably developed by small teams. Version control and configuration management of software are directly related to this element.

i. *Process control*. In software development, the term “process” could be interpreted as the implementation stage of the software development life cycle. Controlling the process thus means establishing procedures for proper monitoring and step-by-step verification of the development of code.

j. *Inspection and testing*. Identification of parameters that should be subjected to testing and documentation of the test procedures and results have to be performed.

k. *Inspection, measuring, and test equipment*. Any test equipment that is used should be periodically calibrated and inspected, and all documents pertaining to these activities should be available. In the software realm, this could be interpreted as ensuring that tools for testing, verification, and measurement are maintained properly.

l. *Inspection and test status*. The test status that a product is in at any point of time should be known.

m. *Control of nonconforming product*. Methods for dealing with defective or nonconforming components should be present. In software development, any defective part is generally reworked thoroughly until it passes all the tests.

n. *Corrective action*. It is not enough to identify and control defective parts; the root cause of the defects needs to be investigated and corrected. This would prevent any recurrence of the defects. Again, procedures must exist to deal with these contingencies. Complete records of the corrective actions have to be maintained to aid future problem rectification.

o. *Handling, storage, packaging, and delivery*. Complete documentation should be maintained for managing the manner in which products are delivered to customers. There should be verification procedures to ensure that the intended product and only the intended product is delivered to the customer.

p. *Quality records*. Once the quality system is in place, it is necessary to maintain records to demonstrate that all procedures are carried out effectively. Identification of the types of records needed for this purpose is essential. Product and process metrics would generally fall under this category.

q. *Internal quality audits*. Periodic review of the entire process by qualified personnel should be carried out according to documented procedures.

r. *Training*. The various levels of skills exhibited by employees and training needs, if any, have to be identified. For every task in the process, the required skill level must be documented.

s. *Servicing*. Procedures for after-sales service for the products have to be chalked out.

t. *Statistical techniques*. The various measures and metrics employed during the process must be validated. Data collection methods and the calculation of metrics have to be tested for accuracy.

In order to ensure that the ISO 9000 standard is interpreted correctly and applied in the appropriate manner to information technology (IT), a registration scheme named TickIT has been developed under the auspices of the TickIT project office of the U.K. Department of Trade and Industry. This has also been supported by the British Computer Society. Under this scheme, auditors are required to follow the TickIt guide that is based on ISO 9000-3, which specifies the guidelines for the application of ISO 9001 to the development, supply, and maintenance of software. This scheme has not yet achieved universal acceptance.

Capability Maturity Model

The CMM provides a layered approach in describing software process maturity. There are five maturity levels that could be correlated to the process management methodology of an organization. Each level has specific process goals, and, if an organization satisfies those goals, it could aim at reaching the subsequent levels of process maturity, thereby working on continuous improvement. The five CMM levels are described as:

a. *Initial*. The software processes are not completely defined. Ad hoc management is performed when sudden crises occur.

b. *Repeatable*. This level is characterized by the existence of process standards that could be employed to repeat earlier types of projects. Infrastructure for predicting business factors, such as cost and schedules, is in place.

c. *Defined*. Organizations in this level have clearly defined, well-documented procedures for process control, and all software projects undertaken by the organization follow these standards.

d. *Managed*. The software process is understood thoroughly, and metrics are collected to constantly monitor and control the quality of the process.

e. *Optimizing*. At this level, the metrics are analyzed and feedback is provided to enhance the quality of the process. New, innovative techniques are employed for continuous improvement.

Organizations that have their process maturity at level 1 (Initial) of the CMM model generally do not possess predictable quality, product functionality, or schedules. Project success, if any, would be highly dependent on individuals, and in their absence the situation would become chaotic. This type of organization typically is crisis-driven and all procedures are ad hoc. Quality control measures are abandoned when customer schedules have to be met.

Organizations in the repeatable level generally have basic software standards defined and followed by the development teams. Earlier successful projects could be repeated because of the discipline in tracking.

In the defined level, software processes are stable and repeatable throughout the organization. Separate process engineering groups exist in some organizations for tailoring any software project to align with the organizational needs.

At the managed level, software quality is considered to be of strategic importance, and measurements are performed to collect and analyze data about the process. A framework for evaluation of any software product or process is present because of the availability of metrics. When standards are followed and all processes are well organized, there is no doubt that products of high quality will be delivered to the customers.

Organizations at the optimizing level are interested not only in high-quality products but in striving for continuous improvement. They employ statistical procedures for understanding the process better.

The maturity level of an organization plays an important role in predicting its capability in meeting demands of cost, schedules, product functionality, and quality. Therefore, organizations have to work toward improving their maturity level incrementally. The awareness of CMM has increased recently among organizations and their customers, and this is a good development from the quality perspective.

The CMM and the ISO 9000 standard have been instrumental in bringing about an immense change in the quality outlook of the software industry. Several factors in the CMM could be correlated to the elements of the ISO standard and vice versa. A considerable amount of overlapping could also be observed in these standards. This is due to the fact that both are fundamentally based on the principle “say what you do and do what you say” (Paulk 1995). Moreover, in both these standards, the emphasis is on monitoring the process and its continuous improvement. While the ISO 9000 standard is applicable to virtually all manufacturing and service industries, the CMM is exclusively a software development standard. Software development organizations should consider adhering to both these standards in their quality plans to achieve wider market acceptance and better process control.

Personal Software Process Model

The PSP, described by Humphrey (1996), is a personal version of the CMM. Indeed, Humphrey believes that for an organization to move beyond CMM level 3, individual staff and team members must implement software process improvement at a personal level. The PSP provides four steps for accomplishing this. In PSP0, a software engineer must learn to measure development time, rate of defect creation, and rate of defect removal. Next, in PSP1, these data are used to estimate the size and development time of new programs. PSP2 emphasizes the significance of focusing on quality from the beginning of a project by requiring the creation and use of planning checklists for design and implementation reviews. Finally, PSP3 provides guidance on repeatedly using the first three steps to allow individuals to scale up to creation of modules several KLOC long. Humphrey believes that one of the keys to industry-wide adoption of the PSP is to include it in the university software engineering curriculum.

5 Software Quality via Better Tools

Industry Practice

The industry leaders in software like AT&T, IBM, Motorola, and Hewlett Packard generally have well-established quality control methods and seem to follow similar procedures, methodologies, and tools in their quality programs (Jones 1994). These companies have invested a substantial amount of time and money on quality management, and their current leadership position in the industry is a direct consequence of this. Their approach is based on proven measurements, methodologies, and tools.

a. *Measurements*. They typically measure defect volume, severity, and origin at every stage of the software development process. These metrics are collected on a daily basis and summarized and reported monthly, quarterly, and annually. They also conduct user satisfaction surveys on an annual basis (Jones 1994).

b. *Methodologies*. They rely on formal inspection of design and code to a great extent. This ensures that when the software product reaches the testing stage, most of the problems have already been detected and corrected. Separate quality assurance groups exist in these industries that have the express purpose of ascertaining product and process quality.

c. *Tools*. Tools for estimating quality, measuring defect rates, planning test procedures, analyzing test results, and predicting reliability are generally used. Statistical analysis is performed, and reports are generated.

d. *Culture*. All employees exude quality consciousness. Training programs are generally conducted to instilll quality consciousness. IBM, in addition to measuring customer satisfaction, also monitors the CUPRIMDSO (Kan, Basili, and Shapiro 1994) satisfaction levels (capability, usability, performance, reliability, installability, maintainability, documentation, service, and overall satisfaction). Hewlett-Packard relies on measuring FURPS (functionality, usability, reliability, performance, and supportability). Up-and-coming companies and those that lag behind the leaders in maintaining sound quality practices could benefit by following in their footsteps.

At Hitachi Software (Onoma and Yamaura 1995), a competitive atmosphere has been created between the software design department and the quality assurance department to cultivate quality consciousness among the employees. The software development process is jointly carried out by three departments, design, quality assurance, and production administration. The design department develops the software and is also responsible for the documentation, cost, schedules, and quality. The quality assurance department runs a quality probe that comprises a small percentage of the complete regular test. Only if satisfactory results are obtained are the products subjected to exhaustive testing. Otherwise, the software product is returned to the design department. Several iterations and frequent feedback may be necessary at times. Such rigorous test practices have brought about a remarkable decrease in system failures.

Program Checking List (PCL) and Quality-Progress Diagrams (QPD) are two of the most important quality control tools used in Hitachi (Onoma and Yamaura 1995). For PCL, test items are identified, based on thorough analysis of the requirement specifications, and test cases are developed for each of them. The expected results are recorded and checked against observed outputs. The advantages of using PCL are:

a. Easy repetition of noncompliant test cases.

b. Monitoring test item quality.

c. Testing and debugging need not be performed by the same person.

d. Statistical data collection. The QPDs deal with the number of PCLs tested and the number of cumulative faults detected per day. The QPD curves indicate the number of untested PCLs, cumulative faults found, and the backlogs of faults.

Defect-casual analysis is another low-cost technique that could be easily implemented to reduce software error rates (Card 1993). This was initially developed at IBM but has been accepted in many organizations. It is more of a common sense method based on the fact that personnel involved in the actual software development process have an intimate knowledge about the problems in software and could suggest methods to avoid future occurrences. The testing department generates reports, and the development team members meet periodically to review and discuss the problems. Another team with greater authority performs follow-up action.

In some cases, the quality techniques employed by the hardware manufacturing industry have been adapted and extended to serve the software development process. An example of this is a model developed to determine the quality levels of several releases of a software product. This is based on Hoadley's Quality

Measurement Plan (Weerahandi and Hausman 1994). The average process quality level and the quality indices for the software could be determined using this methodology.

Motorola has adopted the six-sigma concept to achieve total quality throughout the organization. “Six sigma” is a statistical term that denotes a state of zero defects or as close to zero-defect as is humanly possible (Branthwaite 1994). It roughly translates to 3.4 defects per million or 99.9997 percent perfection. The six fundamental steps to realize six sigma are described here.

a. Identification of the product or service provided to the customer.

b. Identification of the customer.

c. Identification of the requirements of the customer.

d. Definition of the process.

e. Identification of potential errors and elimination of wasted efforts.

f. Measurement and analysis to ensure continuous improvement.

The implementation of the six-sigma concept brought about a tremendous improvement in profits for Motorola during the late 1980s. This was due to reduction in rework costs, nonconformance costs, and warranty repair costs.

CASE Tools and Environments

Computer-aided software engineering (CASE) tools are increasingly being employed by software development teams to facilitate improved requirements management, configuration management, documentation control, project verification, system validation, and process management. Productivity improvements have been reported by several organizations using such tools (Chikofsky and Rubenstein 1988). A study on the productivity perceptions of software engineers also supports these claims (Norman and Nunamaker 1989). However, the principal gain is expected to be in the quality because errors and inconsistencies could be detected early in the life cycle and refinement of specifications could be performed to suitably reflect customer needs. Reliability of a system could also be assessed using these tools even before implementation. The number of CASE tools commercially available for different phases of the software development process is truly mind-boggling. There are workbenches, toolkits, and integrated environments performing modeling, project management, analysis, design, and validation. It is quite a challenging task to identify the exact tool that would be apt for an application.

CASE tools for testing and quality control are available from a number of vendors. SQA Enterprise TestSuite from Software Quality Automation, Inc., is a tool for testing and debugging software that could be used for testing Windows client/server applications. Recently, Microsoft has released Visual Test, another testing/debugging tool (Sarna and Febish 1996). Most tools have the ability to generate test cases and repetitive testing to aid the developers. They also have a record/playback feature to store user commands and input sequences and reenact them. Automated Test Facility (ATF) from Softbridge, Inc., has a feature that enables it to run test programs concurrently on workstations that interact with each other (Wallace 1994). SQLBench (Seque 1998) is a CASE tool that can be used to test design feasibility, for creating benchmark tests, and for evaluation of results. TestGen by Scientific ToolWorks (1996) comprises three tools that aid in all software phases. It has a design review expert assistant that analyzes Ada pseudo-code and prepares reviews, a unit test strategy generator, and a test coverage analyzer.

In addition to discrete CASE tools, there are many software development environments that provide life cycle support for software development. One such product is Software through Pictures, a product from Interactive Development Environments (1996) that includes testing capability. Rational's TestMate is a tool for automatic creation, management, execution, and evaluation of software tests for complex and sophisticated Ada systems. TestMate is a part of Apex, Rational's popular integrated software-engineering environment.

Though there is an obvious need for CASE tools/environments, many organizations use them only in a limited fashion. Some of them abandon usage very soon after implementation. The learning-curve plays a very important role in the acceptance and adoption of such tools (Kemerer 1992). Some software engineering environments extending life cycle support pose greater difficulty mainly due to the enormous amount of learning required for effective utilization.

A complete discussion of software engineering environments is beyond the scope of this report; interested readers may refer to Barstow, Shrobe, and Sandewall (1984); Brown, Earl, and McDermid (1992); and Hünke (1981) for further information.

6 Conclusions

The development of large, complex, mission-critical software systems has long been plagued by schedule delays, budget overruns, and poor quality (Jones 1995). Though severe in nature, these problems are not entirely insurmountable if proper methodologies are practiced by the organization. A host of new standards, models, techniques, and metrics are continuously being invented by dedicated researchers to overcome these problems and improve software quality. Not only should metrics and novel techniques be employed, but they must be verified periodically for accuracy and suitability of purpose. The IEEE Computer Society has published a framework (Vollman 1993, Schneidewind 1993) for the standarization of software metrics called the Software-Quality Metrics Methodology 1061 which provides guidelines for identifying, implementing, and validating the metrics used by an organization. Although a number of quality models are readily available, no one model suits all types of organizations. Therefore, each organization has to develop its own models that are in line with its processes, products, and goals.

Implementation of good quality practices often involves considerable time and effort and requires total commitment from the management and all the people involved. However, it has to be realized that there is really no other alternative but to incorporate quality principles in every part of the organization if success in business is the objective.

References

Arthur, L. J. (1993). *Improving software quality-an insider's guide to TQM*. John Wiley & Sons, New York.

Barnard, J., and Price, A. (1994). “Managing code inspection information,” *IEEE Software* 11(2), 59-69.

This paper describes a system involving nine metrics to monitor, control, and improve the software code inspection process used at AT&T Bell Laboratories. The total noncomment lines of source code per thousand lines of code (KLOC), the average lines of code inspected, the average preparation rate, the average inspection rate, the average effort per KLOC, the average effort per fault detected, the average faults detected per KLOC, the percentage of reinspection, and the defect-removal efficiency are the nine metrics employed. Combinations of these metrics not only facilitate assessment of quality but also aid in determining the status of the inspection process and its effectiveness.

Barstow, D. R., Shrobe, H. E., and Sandewall, E., ed. (1984). *Interactive programming environments*. McGraw-Hill, New York.

Basili, V. R., and Weiss, D. M. (1984). “A methodology for collecting valid software engineering data,” *IEEE Transactions on Software Engineering* SE-10(6), 728-738.

This paper describes effective data collection from a goal-oriented perspective. The methodology involves setting up goals and then generating a list of questions to be answered to acheive those goals. After the questions are formulated, the data categories can be established. The next step involves the design of forms to collect data. Once the data has been collected and validated, analyses could be performed. Several examples and some key lessons learned by following this methodology have been provided.

Boehm, B. (1987). *IEEE Software* 4(5), 84-85.

In a letter to the “Quality Time” section, Boehm has presented his top 10 industrial software metrics. He stresses the importance of code walk-throughs. He also says that fixing a problem after a software product has been delivered to the customer is 100 times more expensive than if it were fixed earlier. The verification of the software performance against the requirements early in the life cycle is crucial.

Branthwaite, D. “Six sigma,” http://mansci1.uwaterloo.ca/~msci604/six_s.html (January 1994).

Six sigma is a quality philosophy adopted by Motorola and many other organizations. The steps involved in the implementation of six sigma quality and the obstacles that could be faced during implementation are explained. The financial implications of this quality methodology on Motorola have also been discussed.

Brooks, F. P., Jr. (1995). *The mythical man-month: essays on software engineering*. Addison-Wesley Publishing Company, Reading, MA.

This is the twentieth anniversary update to the pioneering and still-classic work on software engineering and software project management as applied to large software systems (a field we now refer to as mega-programming). Brooks draws on his considerable experience as leader of the development of OS/360 to provide timeless principles as well as warnings to would-be project leaders. The central thesis of the book is that building large software systems is a fundamentally different process from building small ones (just as building a skyscraper is quite different from building a one-story frame house). He elaborates on team structures, tools, procedures, and policies necessary for success in this field.

Brown, A. W., Earl, A. N., and McDermid, J. A. (1992). *Software engineering environments: automated support for software engineering.* The McGraw-Hill international series in software engineering, McGraw-Hill, London.

Card, D. N. (1993). “Defect-causal analysis drives down error rates,” *IEEE Software* 10(4), 98-100.

Defect-causal analysis (DCA) is an inexpensive technique that was originally developed at IBM to reduce software error rates. This method relies on periodic review meetings between project members. The underlying philosophy is that software developers, who would have the most familiarity with the software, would be the ideal candidates for reviewing the test reports and suggesting measures to prevent errors in the future. Follow-up action is performed by another team with more authority. DCA brings about a better awareness of quality, a strong commitment to the process, and a recognition for measurement of quality. The author also cautions about pitfalls like delays and procrastinations.

Card, D. N. (1994). “Statistical process control for software?” *IEEE Software* 11(3), 95-97.

Many software professionals generally believe that software does not lend itself to measurement. However, statistical process control could still be applied to the software world since the focus is on the development process itself and not on the product. Although absolute ratings could not be associated with any process problem, analysis using statistical methods is bound to throw some light on the general functioning of the development process. Some methods of implementing statistical process control are developing simple process models and then monitoring the performance to determine quality. Good candidates for performance indication are the error-insertion rate in coding and the error-detection rate in testing. Control charts facilitate performance monitoring and indicate the lower and upper control limits. This can be compared with the expected performance. Statistical process control is not always simple to implement. Poor documentation of the process, noncompliance to standards, lack of training, and improper selection of measures are some of the major problems that have to be considered.

Chikofsky, E., and Rubenstein, B. (1988). “CASE: reliability engineering for information systems,” *IEEE Software* 5(2), 11-16.

The advantages of using Computer-Aided Software Engineering (CASE) tools for increasing programmer productivity, improving the cost-effectiveness of the software development process, and for producing reliable, good quality systems are discussed.

Comptroller General of the United States. (1979). “Contracting for computer software development--serious problems require management attention to avoid wasting additional millions,” Report FGMSD-80-4, U.S. General Accounting Office, Washington, DC.

Dromey, R. G. (1996). “Cornering the Chimera,” *IEEE Software* 13(1), 33-43.

A significant shortcoming in many quality models is the disconnect between high-level attributes and low-level software characteristics. More specifically, it is very difficult to say how the measurable properties of software and software development processes affect general features (e.g., reliability and maintainability). The author presents a framework, or metamodel, for building quality models which is intended to help remedy this problem. As examples of his approach, software quality models for implementation, requirements, and design are presented.

Garvin, D. (1984). “What does “product quality” really mean?” *Sloan Management Review*, 25-45.

Quality is viewed by people from different disciplines in different ways. The transcendental approach, the product-based approach, the user-based approach, the manufacturing-based approach, and the value-based approach are the five major approaches to quality that are prevalent. Garvin has also identified eight elements or dimensions that could be used as a framework to determine the quality of a product: performance, features, reliability, conformance, durability, serviceability, aesthetics, and perceived quality. One or more of these elements could be targeted by organizations to achieve increased market gains and reduced cost.

Gibbs, W. W. (1994). "Software's chronic crisis," *Scientific American* 271(3), 86-95.

Yet another discussion of the problems involved in the development of large software systems. It includes the problem-plagued software-controlled baggage handler at the new Denver airport as an example.

Haag, S., Raja, M. K., and Schkade, L. L. (1996). "Quality function deployment usage in software development," *Communications of the ACM* 39(1), 41-50.

Total Quality Management (TQM) comprises three important activities: planning, Quality Function Deployment (QFD), and statistical process control. The authors have adapted and extended QFD that is generally used in the manufacturing environment to suit the software development process and have named it as Software Quality Function Deployment (SQFD). This method focuses on software process quality improvement by implementing novel techniques during the requirements phase of the software life cycle. After the requirements are obtained from the customer, they are translated into technical specifications. A correlation matrix is then completed to detect incompatibilities. The requirement priorities for the customer and the technical specification priorities are then developed. The advantages of using SQFD are discussed in detail.

Halstead, M. H. (1977). *Elements of software science.* Elsevier North-Holland, New York.

The pioneering work in the field of software metrics.

Interactive Development Environments. "Software through Pictures profile," http://www.ide.com/Products/stpsumserv.html (1996).

Humphrey, W. S. (1996). "Using a defined and measured personal software process," *IEEE Software* 13(3), 77-88.

The author argues effectively that improving software quality not only requires improving an organization's development approach à la CMM) but also at the level of individual software engineers. This personal software process (PSP) model is defined and discussed, and some statistical results measuring the PSP maturity of software engineers are presented.

Hünke, H., ed. (1981). *Software engineering environments*. North-Holland, Amsterdam.

International Standards Organization. (1992). *ISO 9126 information technology - software product evaluation - quality characteristics and guidelines for their use*, Geneva, Switzerland.

Jones, T. C. (1994). "Software quality tools, methods used by industry leaders," *Computer* 27(4), 12.

Software industry leaders such as Motorola, AT & T, and Hewlett-Packard employ well-established quality management procedures for their quality assurance programs. They rely on measuring the process parameters regularly and cultivating a quality culture throughout the organization. Defect prevention methods are used to ensure that the majority of faults are detected and corrected well before the software product enters the testing phase.

__________. (1995). "Patterns of large software systems: failure and success," *Computer* 28(3), 86-87.

Typically, large software systems are plagued by schedule delays and budget overruns. Improper and inadequate project management and planning appear to be the primary reasons for such problems. Excessive defect levels and poor reliability even cause cancellation of nearly 20 percent of software projects. Clearly, there is a need for exhaustive procedures to maintain software quality. The deployment of state-of-the-art estimating tools and focus on quality control are some of the recommendations to prevent software disasters.

__________. (1997). "What are function points?" http://www.spr.com/library/0funcmet.htm.

The advantages of using function points (FP) as a metric to measure software productivity are elaborately discussed in this article from Software Productivity Research. An extension of the FP metric called feature points developed by this group is also been described.

Joyce, E. (1987). "Software bugs: a matter of life and liability," *Datamation* 33(10), 88-92.

This article chronicles accidents involving the Therac 25, a radiation therapy device. Embedded software was used in place of mechanical interlocks to prevent high dosages of radiation. After failure of this software resulted in several deaths, hardware interlocks were finally installed.

Kan, S. H., Basili, V. R., and Shapiro, L. N. (1994). "Software quality: an overview from the perspective of total quality management," *IBM Systems Journal* 33(1), 4-18.

This paper describes software quality from the Total Quality Management (TQM) perspective. Irrespective of the manner of implementation, TQM typically comprises the key characteristics, customer focus, process improvement, process measurement, analysis, and human factors like management commitment and total participation. The advent of new technologies like object-oriented development, software engineering environments, and quality models and their effect on the quality of software have also been discussed.

Kemerer, C. F. (1992). "How the learning curve affects CASE tool adoption," *IEEE Software* 9(3), 23-28.

In spite of the obvious need for CASE tools, the acceptance of CASE has been below expectations. Organizations use these tools sparingly or abandon usage very soon after implementation. The learning curve plays an important role in the adoption of CASE technology. The author discusses the necessity for developing learning curve models and their usefulness in estimating the schedules for future projects and better management.

Kemerer, C. F., and Porter, B. S. (1992). "Improving the reliability of function point measurement: an empirical study," *IEEE Transactions on Software Engineering* 18(11), 1011-1020.

To ensure proper control of the software process, measurements have to be performed. The key metric used until a few years ago to measure productivity was the source lines of code (SLOC). An alternative metric has been developed by Albrecht at IBM called the function points (FP) metric. This describes a system in terms of its inputs, outputs, inquiries, and files and has been widely accepted. Though used predominantly as an MIS productivity metric, FP has also been used in cost estimation, software development productivity evaluation, software maintenance evaluation, software quality evaluation, and software project sizing. Researchers have found FP to be a reliable but imperfect metric. Therefore, the authors conducted an empirical study to identify the major sources of variation and establish FP as a more reliable metric. The effects of the variation have also been estimated from case studies conducted using commercial systems. The results of the study have indicated that a small number of factors affect the reliability in a big way.

Kitchenham, B., and Pfleeger, S. L. (1996). "Software quality: the elusive target," *IEEE Software* 13(1), 12-21.

Companies all over the world have invested a lot of resources to find ways to improve the quality of their software. There are several perspectives, viz., the transcendental, user-based, manufacturing, product-based, and value-based, that look at quality from different viewpoints. The measurement of quality has different perspectives as well. Users of software might measure the quality of the product based on reliability and ease of use, understanding, and installation. At the same time, the manufacturer's view is based on defect counts and rework costs. Quality is defined as a hierarchy of factors, criteria, and metrics in the McCall's quality model and as being made up of characteristics like reliability, usability, efficiency, maintainability, and portability in the ISO 9126 model. Both these models look at product quality. Process quality is also of considerable importance, but, above all, an organization's view of quality is based on its business goals.

Magnavox Electronic Systems Company. (1990). *Development software support environment (DSSE) evaluation for Version 1 of the Advanced Field Artillery Tactical Data System (AFATDS)*, Fort Wayne, Indiana.

An evaluation performed by Magnavox of three software development environments for Ada is described. The three were a DEC VAX-based environment, the Rational Environment, and the Army Tactical Command and Control System programming support environment. This study awarded the Rational Environment the highest score.

McCabe, T. J. (1976). “A complexity measure,” *IEEE Transactions on software engineering* SE-2(4), 308-320.

McCabe talks about the development of a mathematical technique based on graph theory to determine if program modules would be potentially difficult to test and maintain. The only metric that was considered important at that time was the number of source lines of code. McCabe has provided a formula for calculating the cyclomatic complexity of code modules. Instead of limiting the physical size of the code, if the complexity of modules is kept below 10, testing and maintenance could be better managed.

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). “Factors in software quality,” AD/A-049-014/015/055, National Technical Information Service, Springfield, VA.

McGregor, J. D., and Korson, T. D. (1994). “Integrated object-oriented testing and development processes,” *Communications of the ACM* 37(9), 59-77.

An integrated approach to software development and testing by employing object-oriented methods is the focus of this paper. The development process and the testing process are not two sequential steps but have to be interwoven. Early detection of faults is possible since testing is performed frequently.

Norman, R. J., and Nunamaker, J. F., Jr. (1989). "CASE productivity perceptions of software engineering professionals," *Communications of the ACM* 32(9), 1102-1108.

The authors conducted a study to determine the productivity perceptions of MIS professionals who use CASE technology. They used a psychometric scaling method called multidimensional scaling to represent the similarities between objects spatially. The subjects of the study were requested to rank pairs of CASE functionalities in terms of effect on productivity. The results indicated that programmers using CASE did perceive productivity improvements. The other observations from the study have also been listed.

Onoma, A. K., and Yamaura, T. (1995). "Practical steps toward quality development," *IEEE Software* 12(5), 68-78.

The quality methods followed by Hitachi Software to reduce system failure levels have been explained in detail in this paper. Program Checking List (PCL), Quality-Progress diagrams (QPD), and quality probes are some of the tools and techniques that have been employed to monitor the process. The advantages of these techniques include easy repetition of noncompliant test cases and statistical data collection.

Paulk, M., Curtis, B., Chrissis, M., and Weber, C. (1993). "Capability maturity model for software, version 1.1," Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

This is the description of the Capability Maturity Model (CMM) which serves to rate the capability of an organization to effectively produce software.

Paulk, M. C. (1995). "How ISO 9001 compares with the CMM," *IEEE Software* 12(1), 74-84.

The similarities and the differences between the ISO 9000 standard developed by the International Organization for Standardization (ISO) and the Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI) are described in this article. The author is one of the developers of the CMM. The ISO standard addresses the minimum requirements for a quality system that would be acceptable. It could be applied to any manufacturing or service industry. But, the CMM is exclusively a software industry standard. Both standards stress the importance of documentation. Some elements of the ISO could be directly mapped to the CMM standard while some others need proper interpretation. Direct correlation between the ISO 9000 standard and the CMM is not always possible. However, organizations need to consider both for their quality process. The CMM is necessary due to its specialized treatment of software, and ISO is necessary to cover a wider market.

Pfleeger, S. L. (1997). “Assessing measurement,” *IEEE Software* 14(2), 25-26.

This is the guest editor's introduction to an issue of IEEE Software devoted to software metrics and measurement. Articles address the current state-of-the-art of software measurement, how to implement an effective software measurement program, software reliability, and the relationship between defect density and software component size.

Rational. (1998). “Rational Apex: Straight anwers to tough questions about team-based development,” White Paper, http://www.rational.com/products/apex/prodinfo/whitepapers/dynamics.jtmpl.

Sarna, D. E. Y., and Febish, G. J. (1996). “Our favorite tools,” *Datamation* 42(4), 24-26.

Computer-aided software engineering tools for the various phases of the software life cycle have been described in this article. The authors have named the best tools for specific tasks.

Schmauch, C. H. (1994). *ISO 9000 for software developers*. ASQC Press, Milwaukee, WI.

Schneidewind, N. F. (1993). “New software quality metrics methodology standard fills measurement need,” *Computer* 26(4), 105-106.

This article describes 1061, the standard for a software quality metrics methodology issued by IEEE. The philosophy behind the standard and its scope for implementing and validating software metrics have been explained. Correlation, tracing, consistency, predictability, discriminative power and reliability are recommended as some of the validation criteria.

Scientific Toolworks. “TestGen—the Ada-based software test tool,” http://www.cyberg8t.com/ssd/testgen.html (1996).

Sethi, R. (1996). *Programming languages: concepts and constructs*. Addison-Wesley, Reading, MA.

This is a standard text on programming languages; the introduction gives the Mariner rocket failure as an example of software failure in the context of this subject.

SIAM News. (1996). “Inquiry board traces Ariane 5 failure to overflow error,” SIAM News 29(8), 1, 12-13.

This is a newspaper-style article which provides details on the inquiry into the rocket/software failure.

Simmons, P. (1996). "Quality outcomes: determining business value," *IEEE Software* 13(1), 25-32.

Nine Australian organizations were surveyed to determine how they measured the effectiveness of information technology projects. The aspect of software quality addressed here was not the quality of code itself but what the software did to improve business. The results indicated that six of the nine believed that measuring this effectiveness was important. All nine generally wanted a quantifiable financial metric, and some even used alternative nonfinancial measures. Based on these results, the author constructed a framework for categorizing benefits according to whether they increased efficiency, increased effectiveness, added value, produced a marketable product, or provided necessary infrastructure for other activities.

Seque. (1998). "Silk performer: e-business load & performance testing," http://www.seque.com/html/s_solutions/silk/s_performer.htm.

Tervonen, I. (1996). "Support for quality-based design and inspection," *IEEE Software* 13(1), 44-54.

The author presents an extension to the software quality metrics (SQM) model which he terms the SQM synthesis model. This extended model is shown to be ISO 9126-compliant. Software tools to aid software engineers in applying the model in an actual development project are discussed.

Ulvila, J. W., and Brown, R. V. (1982). "Decision analysis comes of age," *Harvard Business Review*, 130-141.

The the multi-element component comparison and analysis (MECCA) methodology for computing scores associated with a hierarchy of attributes is described. MECCA works by assigning weights which sum to one to all of the child nodes of each parent node in the attribute hierarchy. After scores are assigned to the leaf nodes, the weights are used to propagate scores to the root.

Vollman, T. E. (1993). "Software quality assessment and standards," *Computer* 26(6), 118-120.

The quality characteristics defined by ISO 9126 serve as an effective framework for evaluating the quality of competing software products. These characteristics have to be refined into a set of attributes that the software products should exhibit. Quality could then be determined by rating each attribute. However, this would be a subjective method. Recently, the IEEE Computer Society has published a standard 1061 (standard for a software quality metrics methodology) that provides a methodology for proper metrics usage. Several approaches have been suggested to objectively measure software. The French National

Standards Body has recommended a list of requirements that all members of a specific class of software should possess for certification.

Wallace, P. (1994). “Testing client/server applications as they're built,” *InfoWorld* 16(21), 90.

Automated tools for testing and debugging software take out the drudgery of repetitive testing. Input sequences could be recorded and played back to verify consistency of performance by the software.

Weerahandi, S., and Hausman, R. E. (1994). “Software quality measurement based on fault-detection data,” *IEEE Transactions on Software Engineering* 20(9), 665-677.

The Quality Measurement Plan (QMP) proposed by Hoadley to determine the quality of hardware during the manufacturing process has been adapted and extended by the authors to develop a parametric model to determine the quality levels of several releases of a software product. Fault detection data are collected after a number of releases of the software product. Analysis of this data using the new methodology allows the authors to estimate the average process quality level and the quality indices for the software.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 1999	**3. REPORT TYPE AND DATES COVERED** Final report

4. TITLE AND SUBTITLE An Introduction to Software Quality	**5. FUNDING NUMBERS**
6. AUTHOR(S) Buvaneswari K. Venkataraman, William A. Ward, Jr.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Faculty Court West 20 School of Computer and Information Sciences University of South Alabama Mobile, Alabama 36688	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Engineer Waterways Experiment Station 3909 Halls Ferry Road Vicksburg, MS 39180-6199	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** Technical Report ITL-99-4

11. SUPPLEMENTARY NOTES
Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

This report is an introduction to the concepts which must be understood and the issues which must be resolved in order to produce quality software. Quality software is defined. Motivation for studying software quality, in the form of examples of past software fiascos, is presented. Various approaches for producing software quality, including better evaluation techniques, better quality measurement, better development processes, and better development tools, are each addressed in separate chapters. An annotated bibliography is provided.

14. SUBJECT TERMS Software metrics Software quality Total quality management	**15. NUMBER OF PAGES** 40
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**