PUBLICATION

Achieving High-Quality Software Systems:
A Comprehensive Approach to
Testing and Validation

Charles Popper

March 2004

Program on Information Resources Policy

Center for Information Policy Research

Harvard University

The Program on Information Resources Policy is jointly sponsored by Harvard University and the Center for Information Policy Research.

Charles Popper is cofounder and CEO of TechPar Group, a technology advisory services company that consults on all aspects of the information technology industry. Previously, he was vice-chairman and chief technologist for Orama Partners, an investment bank serving high-tech start-up companies. From 1991 to 1999, he was chief information officer at Merck and Co., and before joining Merck, he was a partner in the management consulting practice of Deloitte & Touche.

E-mail: pirp@deas.harvard.edu URL: http://www.pirp.harvard.edu
ISBN 1-879716-88-7 **P-04-1**

Acknowledgements

The author gratefully acknowledges the following people who reviewed and commented critically on the draft version of this report. Without their consideration, input, and encouragement, this study could not have been completed:

Bernard Abramson
Alex Berson
David Cosson
Y. Shafee Give'on
John Goodenough
Stephen Greyser
Yu-Chi Ho
Kailash Khanna
John Kneiling
Wolter Lemstra
Peter Maggs
Ian Miller
Bernard Plagman
Cheryl Roby
Ben Shneiderman
Ernest Von Simson
Supriya Singh

These reviewers and the Program's Affiliates, however, are not responsible for or necessarily in agreement with the views expressed here, nor should they be blamed for any errors of fact or interpretation.

I would like to acknowledge in particular the invaluable contributions of Toby Moskovits, who assisted with much of the research and drafting of this report.

I would also like to acknowledge Tescom Software Systems Testing, Ltd., for their support during the preparation of this report.

Executive Summary

In the twenty-first century, when software has become a key force in daily life and its malfunctioning can threaten the public health, safety, and economic well-being, the challenge is to ensure that the quality of software systems is the highest possible. This report presents an approach to analyzing poor quality software systems, by examining their effects, the nature of their defects, and the causes of these defects. A broad theory of quality management is applied to evaluating the quality of software, and broad concepts of total quality management and six sigma are related and applied along with such concepts such as the prevention and detection of defects and estimations of reliability.

Four principles of high-quality software are developed. The first principle is that metrics for the evaluation of quality and other, associated targets need to be defined for each stage of the development life cycle for software. Second, a method for the management of quality is essential to keep the process of improvement going forward. Third, experience, expertise, and training in the planning and use of testing procedures focused on quality also are essential. And, fourth, whether or not a software system meets its requirements should be determined by independent agents—neither the developers, who were paid to produce the systems, nor the technology vendors, who sell off-the-shelf products, have the objectivity required for impartial review.

The report concludes with an analysis of the benefits of a good, comprehensive program for testing the quality of a software program, in particular, the benefits of using independent experts to manage the quality of the program.

Contents

Tables

Chapter One

Introduction

In the twenty-first century, software has become a key force in daily life. Software controls nearly all the devices on which business, education, health systems, and family and recreational activities depend. When these devices work properly, people glide through their daily routines, unaware of the presence of the devices. But when, as happens all too often, a computer-controlled device fails, owing to either a permanent or a transient defect, then, in a markedly unpleasant way, people become aware of their dependence on software. Malfunctioning traffic lights can cause collisions that result in injuries, even fatalities. Improperly programmed medical devices can lead to an incorrect diagnosis or to improper treatment regimens.

For the most part, software glitches are the result of human error. Software systems do not write themselves. The human mind translates a need into functional requirements and useful code. The challenge is to ensure that a product of quality emerges from the process of developing the software.

The first step toward this goal is understanding what quality is. According to the dictionary, quality is "the degree of excellence that a thing possesses." Quality is, undoubtedly, a subjective measure. To define the benchmark of a product's quality requires an understanding of the expected benefit of that product. In producing a product of quality, the goal is to build a product that fulfills the user's "perception of quality." This goal can be achieved by implementing techniques of thorough and nuanced communication and verification throughout the life cycle of the product that will ensure that the desired requirements have been specified, that these requirements are met in the form of agreed-on deliverables, and that the final product meets the user's original expectations. Insufficient communication can undermine the core definition of what the product is to look like in its highest form of quality. If the various stakeholders do not agree on the benefits and requirements of the product, achieving a unified perception of quality among them will prove practically impossible.[1]

Once a software product or system has been built, testing is used to measure the perceived quality. Although there is no objective measure for quality and it is impossible to know the actual quality of a software product, testing can yield an informed assessment of quality.[2]

---

[1]Steven J. DeMarte, "The Quality Perception," *QA Insights White Paper*, [On-line]. URL: http://www.gainsights.com./whitepaper1.pdf.

[2]James Bach, "Satisfice Test Strategy Model," [On-line]. URL: http://www.satisfice.com/tools/satisfice-tsm-4p.pdf.

What, then, is quality software? It needs to fulfill the expectations of users and of rigorous requirements; it needs to be sufficiently free of defects, tolerant during operation of various error conditions, and delivered on time and within budget; and it needs to be maintainable.[3]

Why is such quality so hard to achieve? Software systems have mushroomed from thousands to hundreds of millions of lines of code and, along with this complexity, systems have become error-laden. As of this writing, 80 percent of the costs of software development go toward identifying and correcting "bugs," yet the systems produced still have more defects than any other products sold today.[4] One key aspect of the poor quality of software is the effect of thorough testing and management on the process of producing software. This report proposes a set of management principles that can contribute to the development of software of high quality.

In Chapter Two, the problem is defined, its effect is described, and concepts of quality management are reviewed. In Chapter Three, how software defects occur and how they can be detected and corrected are analyzed, along with the costs of correcting software defects and the issue of software quality in regulated environments. Although the focus is on management and testing for quality, the concept of "designing for testability" is also addressed (the broader topic of sound design principles is beyond the scope of this report).

In Chapter Four, four principles of management for quality are presented that support the development of high-quality software, and their potential benefit is described. These principles were developed specifically for application to the quality of software, and their use will go a long way toward the development of software of better quality. The application of these principles to an existing framework, such as total quality management or a capability maturity model, can lead to even better results.

Chapter Five is a discussion of the use and value of independent testing along with a case study of the use of independent quality management. Chapter Six summarizes the benefits of the quality programs discussed in this report.

---

[3]Software QA Testing Resource Center.

[4]Press release, NIST, "Software Errors Cost U.S. Economy $59.5 Billion Annually," (June 28, 2002), [On-line]. URL: http://www.nist.gov/public_affairs/releases/n02-10.htm. For the full report, see URL: http://www.fda.gov/cdrh/ode/ssoftware.pdf.

Chapter Two

Background: Definition of Problems and Current Concepts of Quality

2.1 Analyzing Systems of Poor Quality

The first step toward solving a problem is to comprehend the problem and how it is manifested in the real world. Achieving software of high quality requires a thorough understanding of the ways that current software systems fail.

2.1.1 Systems That Do Not Satisfy Users' Functional Requirements

A poor quality system is one that does not satisfy its users' requirements. The requirements define the function that is to be built into a product, including suitability, accuracy, and interoperability, as well as other nonfunctional requirements. The nonfunctional requirements include security, privacy, performance, modifiability, and compliance with relevant regulations and standards in the product's domain (**sections 2.1.4** to **2.1.6**).

The specification of a product's functional requirements is the itemization of what the product will do, that is, the product's operational features. The essential aspect of the quality is the capacity of its operational functionality. A system that does not fulfill its users' expectation that the product will be useful and its process accurate for a particular task will be perceived as a product of poor quality. Equally important is the need for it to be interoperable with existing operating systems, relevant software products, and hardware components.

An important challenge in testing functionality is to overcome the tendency to test features in isolation, which often may mean ignoring the interactions among features. This is a problem, because even though a product is used for a particular task, its functions are not isolated from other functions. To simulate real-world use, testers need to use task-centric testing.

2.1.2 Unreliable Systems

A key issue with regard to the quality of software is the system's reliability. A reliable system performs its functions in the environments for which its use was specified. Reliability depends on how the system handles errors, its tolerance of faults, its scalability, and its ability to recover. Other aspects of reliability include the ability to counteract bugs or deviations, prevent episodes of failure, or recover from a failure even when the workload exceeds initial expectations. A system that does not resist failure and cannot easily recover from errors or failures is not a system of high quality.

For example, the Food and Drug Administration (FDA) recommends both risk analysis for software for medical devices and control methods that are focused on mitigating the risks in a specific product. The analysis and control should be performed for the entire device as well as for

its major subsystems and components—that is, the focus should be on software, hardware, and biomedical materials. If a hazard in a medical device is linked to its software, the likelihood of an adverse occurrence will be directly related to the failure rate of the software. Risk-reduction techniques need to be used to control the severity of adverse events, among them building systems that will resist failure and have the ability to manage or, at least, recover from failures.[1]

2.1.3 Systems Difficult to Use

The developer organization, which is building the system, needs to identify the end-users of the product and all the relevant stakeholders. A high-quality product comprises a set of features that can be easily understood, an operation that can be mastered rapidly, and operation that requires minimal effort of the end-user. A system that is difficult to use poses several problems: uncertain feedback after the user has entered input, ambiguous prompts, unreasonable demands on the user's mental calculations, language or acronyms that are unfamiliar, no capacity to query for critical inputs, and illogical or cumbersome control or input sequences.[2] A system that poses these problems lacks usability and will be perceived by the end-user as a product of a low quality.

Usability is particularly important for software used in medical devices. Among the failures documented by the FDA was a radiation device that functioned improperly because the user failed to input a dosage. Because the software did not query the user, no default value was displayed and no warning alarm sounded. Another documented case concerned a monitor for use with newborns that did not sound an alarm for a very high heart rate that increased beyond a certain level of measure; as a result, the patient was injured and required emergency care.[3]

2.1.4 Inefficient Systems

Performance includes the speed at which the software carries out its tasks and responds to the user's commands or requests as well as the ability of the software to use resources, such as computer processors, internal memory, disk storage, communications devices, and so on. A system that lacks the ability to provide scalable performance will not work well when it is moved from the realm of development and testing into a real-world environment, where load is an issue. Sometimes performance can be improved by adding more resources, but providing them affects the operational cost of the system and may render its continued use uneconomical. In the worst case, it may be impossible to provide the necessary additional resources, and the system will need to be replaced or substantially redesigned and modified.

---

[1]United States Dept. of Health and Human Services and Food and Drug Administration, "Guidance for FDA Reviewers and Industry: Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices" (May 29, 1998), 18, [On-line]. URL: http://www.fda.gov/cdrh/ode/software.pdf.

[2]Ibid., 26-27.

[3]Ibid., 27.

2.1.5 Systems Lacking Proper Security

It is perhaps an understatement to say that today's world offers many security challenges, and an important aspect of a system's quality is its ability to provide adequate security. There are many levels of security and there are specific security objectives. For example, financial systems need to protect the integrity and confidentiality of the transactions they are processing and of the accounts and identities of the parties making the transactions. Many systems need to provide a "nonrepudiation" mechanism, so that the legitimacy of a transaction cannot later be denied by the person who entered it. Industrial and commercial systems need to protect themselves from attacks that could limit their availability. Military systems are even more sensitive in this regard.

Developing software systems for regulated industries necessarily involves dealing with issues of regulatory compliance. Legislation such as the Health Insurance Portability and Accountability Act of 1996 (HIPAA) may raise critical security and privacy requirements. Without password protection and well-designed, well-tested encryption algorithms, a system could be vulnerable to inappropriate and unauthorized access to records and illegal modification of sensitive data, such as patient reports and clinical data.

2.1.6 Systems Difficult and Costly to Maintain

The ability to maintain a software system easily is important to the quality of the overall system, because all systems undergo significant change during their life cycles. Within six months of their initial release, many systems require a "release 2" to provide functionality missing from the first release or to correct defects that arise when users find it difficult to envision or express clearly what they want in the requirements. The maintainability of a system can be affected by many factors—high-level architecture and design, adherence to standards, the stability and quality of the development team, among others.

From the perspective of quality management, a system can be more easily maintained if it is properly documented, both at the stage of the requirements analysis and throughout the process of development. Documentation is a key to the ability to analyze, maintain, and alter existing systems, as required. Too often, however, developers build systems as if reexamining the code or the processes will never be necessary and in this way create substantially more work for themselves (or their successors), when maintenance or the development of a later version is needed.

Testability is central to quality. If a system cannot be tested, there is no final assurance of its quality before the customer uses the system. The proper requirements analysis and documentation are vital for good testing to take place, as is the implementation of an iterative process that is woven through the development life cycle. Because these allow early detection of bugs in the software, testing has a role in the attainment of quality software, rather than being simply a tool used to measure quality. Equally important is a product's stability during development, testing, and its subsequent customer use.

2.1.7 Systems Difficult to Interface with or Port to New Environments

In real life, systems need to evolve. External factors may dictate that the system will be moved (ported) to a new environment, and new applications of the system may impose a need for an interface with new systems or devices. A product of high quality will easily be ported to new environments, and its technology and architecture will be usable—or reusable—there, without any major need for a "rewrite" or redesign that might undermine the quality of the system. The adaptability of the technology, ease of installation on new platforms, conformance with existing standards, and the ability to replace components with new or improved systems all are vital. The ability of a system to interface with existing or future software systems and to work compatibly with external components and configurations is of great importance. Systems that lack such capabilities when circumstances require them are systems of poor quality. Portability and the ability to interface easily with other systems are particularly important in relation to Web services, which are a growing information-processing paradigm, in which processing components need to interoperate dynamically. A well-behaved Web service will interoperate successfully with all other Web services, in combinations that cannot be tested in advance of performance.

The issue of the interface is particularly important, for example, in software for medical devices. All external accessories—including software to use in planning radiation treatment, drug library editors for infusion pumps, and software to analyze the results of electroencephalographic testing—are governed by the same quality requirements as the primary software system. These external accessories include also components that accept data from users and modify them for input into a medical device and components that accept data from the device and present them to the user. The ease with which this interface is accomplished is vital to the functioning of the device and to the system's overall quality.[4]

An example of the importance of portability is a security product, such as for encryption or authentication. Such products run in many environments, but these may not all be known when the products are initially developed. For this reason, the design and implementation of the product need to be capable of future implementations in new environments with the use of techniques that straightforward and will not require changes to the installed base for compatibility and interoperability.

2.2 Economic and Human Effects of Systems of Poor Quality

Studies conducted in the past few years examined the economic effects of the failure of software systems. A study commissioned by the National Institute of Standards and Technology of the U.S. Department of Commerce found that software bugs cost the economy approximately $59.5 billion annually,[5] or the equivalent of 0.6 percent of the U.S. gross domestic product.

---

[4]Ibid., 30-31.

[5]Ibid., 26-27.

Ultimately, these costs are distributed to software vendors and users, with roughly 60 percent of them borne by the user. The same study found that although all errors cannot be removed from a software system, an estimated $22.2 billion, or a third of the cost of the errors, could be eliminated if the developers' testing infrastructure were improved to allow them to be identified earlier which would make correction more effective. The savings would be achieved by uncovering errors closer to the point of the system's introduction into use, thus minimizing the complexity of removing the errors and the effort required to remove them. In most environments in which software is developed, more than 50 percent of errors are discovered late in the development cycle or often not until the software has been deployed.

Financial losses occur when faulty software is deployed. In 1999, for example, the Hershey Food Corporation deployed a $112 million order-management-and-distribution platform, which had been developed over a period of two and a half years. When the system began to delay orders, Hershey could not make crucial deliveries of Halloween candy and lost an estimated $120 million in sales.[6]

Such substantial financial losses pale by comparison with the potential for damage caused by bugs or errors in a system used in a regulated environment such as the pharmaceutical industry or the medical-device industry. There such software glitches or accidents within systems that consist of complex interactions between components and activities can lead to injury and death. One widely noted accident related to software affected the use of a computerized radiation therapy machine, the Therac-25, a medical linear accelerator that was designed to deliver high-energy beams that could destroy tumors with minimal effect on the surrounding healthy tissue. Between June 1985 and January 1987, six accidents were found to have occurred that were connected with the use of this machine and that resulted in injury and death. The Therac-25, unlike earlier versions of this technology, required less space to reach comparable energy levels, and, unlike previous models, it was not only more compact but also easier to use. Unlike earlier systems, it was software-centric—the device was designed to take full advantage of computerized controls for maintaining safety, and the safety interlocks and mechanisms were software-based and not duplicated in the hardware. The user interface, however, proved not to be foolproof. It permitted operation in modes that were dangerous to the patient.

In the six accidents, massive overdoses of radiation were delivered during the treatment of patients. In the first accident, in 1985, at the Kennestone Regional Oncology Center, in Marietta Georgia, burns developed during follow-up treatment with radiation in a 61-year-old patient who had undergone a lumpectomy. The patient felt a burning sensation and immediately reported it to the radiology technician. On examination, her breast and arm did not show external signs of a burn, but an inquiry after the treatment found that the patient had suffered a serious radiation burn

---

[6]Aaron Ricadela, "The State of Software Quality," *Information Week* (May 21, 2001), [On-line]. URL: http://www.informationweek.com/838/quality.htm.

that required surgery to remove her breast and resulted in the complete loss of use of her shoulder and arm on the affected side.[7] Better quality control of the interlocks and the user interface might have prevented this outcome.

In the manufacture of pharmaceuticals, errors in software that can make the manufacturing process unreliable can result in the spoiled manufacture of batches of precious chemicals as well as heavy financial losses. There are long-term consequences when a company cannot fulfill orders and customers are alienated and, if alternative products are available, they choose to use them, as well as immediate economic effects. There is also always the risk that an error that goes undetected will put faulty chemicals or drugs into circulation. Because core manufacturing processes generally are actively monitored for quality control, what is more liable to occur is a that a defect in the software that controls the documentation and labeling of the product would allow, for example, an incorrect expiration date of a drug to be entered, so that patients would receive drugs that were no longer effective. The potential outcome could be serious illness and death.

The failure of industries regulated by the FDA to meet that agency's quality standards can result in warning letters, fines, the closing of plants, and delays in approval of important new products. Failures resulting from inadequate or inappropriate techniques of software development could lead to the delivery of unsafe drugs or life support. When information about a product is incomplete, what is missing can lead to life-threatening situations such as misdiagnosis and the selection of an inappropriate treatment.

The severe human and economic effects caused by defective software are real. They can happen to any organization, and to any product. Because software design is fundamentally a human activity, there can be no guarantee that it will be error free.

2.3 Related Concepts

A rigorous discussion of software quality requires those discussing it to be familiar with both software development and the quality of software as specific disciplines and knowledge domains.[8]

2.3.1 The Software Development Life Cycle (SDLC)

The system development life cycle, known also as the software development life cycle, is the sequence of phases that any system goes through, from the initial planning activities (concept phase), through developmental phases (specification, design, implementation, which may be

---

[7]Nancy Levenson and Clark S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Comp*. **26**, 7 (July 1993), 18-41, [On-line]. URL: http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.htm.

[8]A review of these areas in depth is beyond the scope of this report.

called the coding or construction phases), to integration, and the final operation of the product. Although there are other names for these phases, there is general agreement about the content of each phase.

An understanding of the software development life cycle is critical to the discussion of software quality, and, because the issue of quality arises in any and all stages of the cycle, appropriate testing and other quality evaluation procedures are necessary at each stage. This life cycle is applicable both to custom-developed software and to the implementation of commercial, off-the-shelf software applications. The targeting and customization of a package to a specific business situation need to be specified, designed, implemented, and integrated. The work products of each of these phases need to be tested and evaluated for quality.

2.3.2 Quality Assurance (QA)

The discipline of quality assurance can be applied to many domains. For software, quality assurance is applied to the development of the software. In general, quality assurance comprises the methods and procedures used to enforce internal rigor in the quality of the processes—by the monitoring, control, review, evaluation, and regulation of the stream of deliverables produced in these processes The chief role of software quality assurance is to act as an internal watchdog.[9] The focus of quality assurance is to make sure that work products adhere to the relevant standards; quality assurance allows the results of the process to be reported to the participants, their managers, and, often, their customers.

2.3.3 Total Quality Management (TQM)

Total quality management is an all-encompassing management philosophy that was developed by W. Edwards Deming[10] in the early 1990s. At its core is a focus on continuous improvement of the process. Total quality management is achieved in many steps[11] that are taken primarily in the realm of organizational culture. For example, a statement of purpose needs to be created and published; a new philosophy needs to be learned and adopted; leadership needs to be adopted and instituted; fear needs to be driven out; trust and a climate in which innovation can occur both need to be created; barriers need to be broken down; teams working toward the aims and goals of the overall enterprise or project need to be optimized; slogans, exhortations, and targets for the work force need to be eliminated; numerical quotas for the work force need to be eliminated; dependence on inspections to achieve quality need to cease; the practice of awarding business on the sole basis of price need to be ended; barriers that rob workers of their pride in

---

[9]R. O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992), 282.

[10]See, for example, URL: http://www.deming.org/.

[11]Lewis, 285.

their workmanship need to be removed; training need to be instituted; education and self-improvement for everyone need to be encouraged; and, most important, the system of production and service need to be improved, both constantly and forever.

The successes achieved with the use of total quality management are recommendations for applying it elsewhere, such as in the production of software. The last point—the need for constant improvement to the process of production—is crucial. There are five steps in the process: thoroughly understanding the product requirements, assessing the product against those needs, understanding what makes a process less than perfect, improving the process to eliminate these problems, and repeating these steps, again and again.

2.3.4 Six Sigma

Six sigma is a vision of quality in which quality is equated with the achievement of a rate of no more than 3.4 defects per million opportunities for a product or service transaction. An "opportunity" is defined as a chance for nonconformance or, put another way, for not meeting required specifications. To achieve six sigma means executing key processes nearly flawlessly. Six sigma represents a goal that can be achieved with the application of various techniques, some of which are described in the next sections.

Design for six sigma (DFSS) is a systematic method that requires tools, training, and measurements to bring the design of products and processes up to the customers' expectations and to produce the design at the level of six sigma quality (**section 3.2.4**).

2.3.5 ISO 9000

The International Organization for Standardization (ISO), which is located in Switzerland and has members from more than 120 national standards bodies, was established in 1947 to develop international standards in many areas.[12] ISO 9000 is a set of quality management standards that currently include three quality standards—ISO 9000:2000, which consists of requirements, and ISO 9001:2000 and ISO 9004:2000, which consist of guidelines. All three are sets of process (not product) standards.

The ISO 9000:2000 standards apply to organizations, including software development organizations, that need to develop a system of quality management to meet the new ISO 9000 standards. The purpose of the ISO 9000:2000 standards is to control and improve the quality of products and services, reduce costs associated with poor quality, and help an organization to become more competitive. Organizations may choose this path simply because customers expect them to do so or because a government body has made the choice mandatory.

---

[12]For information about the ISO and its quality standards, see the Praxiom Co. Web site, at URL: http://www.praxiom.com/. See also the ISO, at URL: http://www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.

Implementation of the ISO 9000 family of standards begins with a gap analysis—that is, an analysis to identify the gap between the new ISO 9001:2000 standard and an organization's processes. After the analysis, steps can then be taken to fill these gaps, in order to comply with the new ISO 9001:2000 standard, and also to improve the overall performance of the organization's processes. Auditing and registration of ISO 9000 compliance can then ensue.

ISO 9000 is important because it is oriented to systems. An ISO 9000-based quality management system will institutionalize the correct policies, procedures, records, technologies, resources, and structures to lead to the development of a culture and attitudes that will support achievement of world-class quality.

2.3.6 Verification and Validation

There are dictionary definitions of the words "verification" and "validation," but in relation to systems that need to meet specific government standards, such as those of the U.S. Department of Defense, the FDA, and the Federal Aviation Administration (FAA), the following definitions are used in this report.[13]

Verification is an iterative process that is aimed at determining whether the product of each step in the development cycle of software fulfills all the requirements required of it by the previous step and is internally complete, consistent, and correct enough to support the step. In the language of the FDA, verification comes under the rubric of validation. Validation, strictly speaking, is the process of running the software on the designated hardware and then comparing test results to the performance that is required, in order to establish objective evidence that the design specifications and their implementation conform to user needs and to the intended use of the software (for the FDA's requirements for validation, see **section 3.5**).

2.3.7 Common Criteria for Secure Computer Systems

The security of an information system is a critical aspect of its quality. There is an agreed-on framework for evaluating security, called the common criteria.[14] These criteria both define general concepts and principles for evaluating the security of information technology (IT) and present a general model to use for evaluation. The criteria include constructs for expressing the security objectives of IT, selecting and defining the security requirements for IT, and writing high-level specifications for products and systems. The common criteria address security functional requirements by establishing a set of components that express in a standard way what the requirements are for targets of evaluation. They also deal with assurance requirements by establishing a set of components that express in a standard way what the security requirements are for the targets of evaluation (the systems being evaluated for security).

---

[13]Lewis, 7-8; see also http://www.iso.com.

[14]URL: http://www.commoncriteria.org.

2.3.8 Capability Maturity Model (CMM)

The capability maturity model® was developed by the Software Engineering Institute to characterize and measure rigorously the ability of an organization to generate high-quality systems. The model is based on work by Watts Humphrey at the International Business Machine Corporation, who was trying to adapt and implement Deming's "continuous improvement cycle" (**section 2.3.3**) for software development. The fundamental insight of the capability maturity model is that software development organizations generally have many impediments to continuous improvement that need to be systematically removed. The capability maturity model represents how organizations develop and change so that development practices can be transformed from an ad-hoc, undisciplined state into disciplined processes capable of predictable results.

The capability maturity model defines five levels of maturity at which the implementation and institutionalization of several clusters of practices (process areas) occur that contribute to the development capability at each level (see **Table 2-1**).

2.3.8 Team Software Process (TSP)

Although the capability maturity model provides a powerful framework for improvement, its focus is necessarily on what organizations should do, not on how they should do it. Many organizations that used the capability maturity model had difficulty applying its principles. The personal software process (PSP) and the team software process (TSP) provide a road map for organizations and persons to use to achieve high performance. The PSP provides guidance for how individual engineers can improve their performance; the TSP extends and refines the capability and the methods of the PSP to guide engineers working in development and maintenance teams. The model and the methods show engineers how to build a self-directed team and how each person can perform as an effective team member. They also show management how to guide and support the teams and how to maintain an environment that fosters high performance by a team. The principal benefit of the TSP is that it shows engineers how to produce quality products for planned costs and on aggressive schedules by showing them how to manage their work and by making them owners of their plans and processes.[15]

In summary, the various approaches to the quality of software have had some positive effects, but the problem of ensuring quality remains unsolved. There are many reasons that this is the case—technical, cultural, managerial, and educational.[16] The approach here to the quality of

---

[15]URL: http://www.sei.cmu.edu/tsp/; see also Watts Humphrey, "Pathways to Process Maturity: The Personal Software Process and Team Software Process," [On-line]. URL: http://interactive.sei.cmu.edu/Features/1999/June/Background/Background.jun99.htm.

[16]This subject is beyond the scope of this report.

software is from the perspective of management and testing, and next focuses on defects in software.

Table 2-1

The Capability Maturity Model

Maturity Level	Process Focus	Characterization	Improvements Implemented
5. Optimizing	Change management	Continuously improving practices	Develop change infrastructure Evaluate and deploy improvements Eliminate causes of defects
4. Quantitatively managed	Capability management	Quantitative under-standing and control	Manage processes quantitatively Establish capability baselines
3. Defined	Process management	Common engineering processes	Establish improvement infrastructure Identify required software processes Define common software processes Deploy and manage processes Collect process-level data Provide organizationwide training Coordinate with nonsoftware groups
2. Repeatable	Project management	Repeatable practices	Manage requirements Plan and track projects Manage suppliers Manage product configurations Measure projects Assist and assure policy compliance
1. Initial	None		

Chapter Three

Analysis: Detecting and Correcting Defects in Software

3.1 Understanding Defects

A common theme in the discussion of approaches to quality in software in the previous chapter was the identification, introduction, and instituting of development processes that will improve the quality of the products at each stage in the life cycle of development. This chapter looks at this theme from the other side, by characterizing the defects that reduce the quality of a software system. Understanding the categories of defects and the ways in which these defects arise means understanding the ways in which different methods of ensuring quality can be applied to prevent defects from occurring in the first place and to locate and eliminate defects after they have been introduced.

The terms error, fault, and failure are often used interchangeably, but their meanings are different. An error generally signifies an action or an omission by a programmer that causes a fault. A fault is a defect that causes a failure. A failure is the departure of a program operation from its requirements. The term defect is used here mean any one of these.

The problem of defects in software is looked at holistically here, with several objectives: to measure the quality of the work at each stage in the life cycle of software development; to identify and mitigate potential problems or deficiencies before they can be introduced into the design or code; to ensure that at each stage the desired quality has been achieved before the next stage has begun. The solution to the problem of defects is not to identify and remove the bugs from the software system after the fact. Because formally proving that programs are correct has been shown to be an intractable problem, whether or not the last bug has been eradicated cannot be known.[1] Many types of problem or deficiency can affect programs (see **Table 3-1**). More examples or permutations could be included than those mentioned here, of course, but the point of the list in the table is that defects can be generated in the software at any point its life cycle—during specification, design, development, implementation, or integration—and a variety of possible causes may lead to the system's failure to function correctly. Each phase in the life cycle requires quality control and the use of appropriate methods of testing (see **Chapter Four**).

---

[1]URL: http://www-cse.stanford.edu/classes/cs103a/h28ProgProof.htm. Readers interested in the theory behind this assertion are referred to a discussion from the perspective of complexity theory of the theoretical impossibility of proving software is bug-free and secure. See Yu-Chi Ho, Qian-Chuan Zhao, and David L. Pepyne, "The No Free Lunch Theorems, Complexity, and Security," *IEEE Trans. Automatic Control* **48**, 5, 783-793 (May 2003).

Table 3-1

Problems and Deficiencies That Can Affect Software Development

The stated functional requirements may not reflect what users of the system desire or need. In this case, the program code may correctly implement the requirement, but the written requirement is faulty.
The system performs all functions correctly but takes too long to do so or uses too many resources, so that the characteristics of performance are unacceptable.
The system performs all functions correctly but it is not intuitive, and poor ergonomics reduce its effectiveness, leading to excessive user errors and poor business productivity.
Logical errors in the system architecture that defines relationships among the components (e.g., how they share data or processing steps), so that, although individual components operate correctly, the system as a whole fails to comply with the functional requirements.
Flaws may occur in the program code of a specific component, because the programmer misunderstood the requirement: the programmer may have correctly understood the requirements but misunderstood the technical details of the system or the programming language; the programmer may have failed to account for all combinations of situations and inputs that the program may encounter; or programmers did not code instructions accurately, as they had intended.

3.2 Detecting Defects, Estimating Reliability, and Instituting Preventative Action

Dynamic software testing is the process of exercising a software system to make a determination regarding specification fulfillment and performance in the environment in which the software is intended to be used. The specification is crucial to this process, because the correct behavior of the software needs to be defined clearly so that incorrect behavior—failure of the software because of faults in the source code—can be identified. If failure should occur, the code developer needs to diagnose its cause by comparing the actual output with the specified and therefore expected output. The limitation of dynamic testing is that the code needs to have been written before the testing can be performed. Static testing can begin during the requirements phase, and thus it complements dynamic testing. Static testing comprises such techniques as conference-room pilots,[2] design walk-through, and use-case analysis,[3] to ensure the quality of the requirements specification and design, even before coding begins.

Testing activities can be directed toward three distinct but interrelated goals: defect detection, reliability estimation and preventive action, which are discussed in the next three sections.

---

[2]A conference room pilot is an event, conducted like a seminar, at which business users and designers of the system to be developed meet to develop the models for the business processes that will use the intended software, evaluate the software functionality, identify software gaps, and refine the configuration of the software.

[3]Use cases are sequences of user interactions with the software system which, taken together, represent all the ways in which ordinary users are expected to use the system. Use-case analysis is the process of determining the set of use cases involved, so that a test program can be designed to exercise them.

3.2.1 Detecting Defects

The detection of defects can be divided into three general categories on the basis of the point of detection: early detection of defects, process defects, and product defects. In early detection, errors are detected before a new system build takes place—that is, during unit and module testing. Defects detected early are the ones most easily corrected, because their effect is more isolated than that of defects detected later.

Defects detected between the point of a new system build and the delivery of the system to the customer—that is, during integration testing—are called process defects. Process defects require a longer correction cycle than defects that are detected earlier. The faulty unit(s) or module(s) need to be corrected and retested; then a new system build needs to be performed, and the integration test repeated.

Defects that are not detected until after delivery to the customer are product defects.[4] They generally require a two-step correction. First, a patch needs to be prepared that can be communicated and made available to customers, who usually download it from an Internet site—a process that creates some potential vulnerability from exposure. Second, the underlying modules need to be repaired for the next full or interim ("service pack") release of the software.

3.2.2 Estimating the Reliability of the Software System

In most software development departments, fault detection and defect detection are the norm, but, because exhaustive testing generally is not feasible, detection testing usually notes only the presence of flaws but not their absence.[5] The goal of a testing program is not merely to eliminate bugs, because one can never be sure that the last bug has indeed been eradicated, but, rather, to characterize accurately the quality and reliability of the system being developed. The system can be accurately characterized through the use of a comprehensive testing program that is applied to all stages of the life cycle of software development.

Estimating the reliability of the software means estimating the probability that the software will provide failure-free operation in a particular environment for a specific period of time. The common measure of reliability is the mean time to failure, that is, the average interval between failures. The probability of failure is defined as the probability of a software failure on the next input event. These two measures can be related by determining the frequency with which inputs are executed.

---

[4]J. Dennis Lawrence and Warren L. Persons [preparers], "Survey of Industry Methods for Producing Highly Reliable Software," U.S. Nuclear Regulatory Commission, Fission, and Energy Systems Safety Program, Lawrence Livermore National Laboratory (Pub. No. NUREG CR-6278UCRL-ID-117524, Aug. 29, 1994), 12-14.

[5]"Definition of Software Testing," [On-line]. URL: www.cigitallabs.com/resources/definitions/software_testing.html.

A key challenge in estimating the reliability of software is to develop accurate operational profiles that will represent the inputs a system will encounter during its lifetime. The profiles may be defined and collected for systems that operate in specific industries or those with long histories, although with some difficulty. For most shrink-wrapped software products, operational profiles are almost impossible to guess. How a variety of users who are using a range of products, each with its own unique features, will behave cannot be known. Some uses might depend on the user's level of expertise, professional responsibility, or personal preferences. To make matters worse, failures do not have an equal effect. Some failures are catastrophic and lead to death or injury, whereas others are more benign, such as misspelled words in system messages. Without a set of good operational profiles, it is difficult to ensure that the testing program will address a large enough proportion of the system's intended operations.

From a mathematical perspective, the precision of a reliability estimate depends on the number of test cases that are run. The tradeoff is between the value of obtaining a more precise estimate and the cost of a large testing program. Simply increasing the number of tests may not improve precision, if the additional tests fail to exercise additional parts of the operational profile. An alternate method is to compute the convergence of the testing process, by comparing the rates at which new bugs are discovered and the old bugs are repaired. If the tests cover the operational profiles adequately, convergence—shrinking the total number of the remaining bugs—can be used as a surrogate measure of the reliability.

Owing to limitations on time and resources in most testing organizations, the division of time and resources to be used for defect detection and estimating reliability becomes an important consideration. Before resources can be allocated for the detection of bugs, the testing team needs to attempt to perform an analysis to estimate the quality, or the degree of error, in the product. The analysis is intended to reduce uncertainty and enable the product managers to make decisions about allocating resources to remove defects, include or remove other features, and about the timing of the initial and subsequent releases of the software.

3.2.3 Preventive Action

Also of importance in planning testing is determining the appropriate point at which to begin testing. When testing is viewed as part of improving quality after the fact, it becomes the last defense before the developer makes the system available to the end-user. At that point, the tester is forced into the position of an assessor of quality, rather than a facilitator of it, because time is short and real changes are too costly. For testing to have a useful effect, the designing of testing needs to begin before the coding. In addition to preventing bugs, early designing of testing

permits the identification of problems in the user interface and in usability, when adjustments are more economical.[6]

When the planning of testing, analysis, and design takes place early in development (another aspect of static testing), they have been shown drastically to improve the quality of software specifications, because using software requirements at this point can generate questions that will reveal any ambiguity and incompleteness, when changes and corrections are relatively cheap. Software specifications and code can be refined by their use early in testing. Testers can build models to show the consequences of the specifications and thereby draw the attention of analysts, developers, salespeople, and customers to the consequences of the design and requirements decisions. Use of these models early in the development process allows key stakeholders to recognize the need for changes in the design and requirements when it is economical to make them.[7] When information on test design is available early, it can be useful in design inspections and in testing by the developer.

Preventive action, along with detecting defects and estimating reliability, is crucial to establishing a quality-focused organization that has high-quality development processes. Preventive action can provide quantitative feedback for a concerted, lasting focus on process improvement. To ensure the existence of such organizational practices, the testing professionals within the company need to make sure that the relevant data and measures are transmitted back to the development professionals, who, when they see the unintended consequences of their activities, will learn to improve these activities and avoid comparable mistakes. To this end, detailed measures, analysis, and reporting are important. User feedback when the product is in the field, which allows testers and developers to see whether the reliability estimation is accurate, is also important. Preventative action is key to achieving software of good quality and needs to be part of every aspect of the process of its development.

3.2.4 Designing for Testability

A number of design techniques contribute to making software testable. One is the division of a system into modules, each of which has a well-defined task and well-specified inputs and outputs. Modules are easily tested, because it is not difficult to develop a set of test data that exercise all the module's functionality. With the increasing complexity and difficulty of the module, the designers and testers will be less likely to understand it and the test cases will be less likely to cover expected uses of the system, with the result that the tests may fail to exercise the system's defects. The challenge in designing for testability is to maintain control of the

---

[6]Brian Marick, "Classic Testing Mistakes," (1997), Testing Foundations, [On-line]. URL: http://www.testing.com/writings/classic/mistakes.html.

[7]David Gelperin and Bill Hetzel, "The Growth of Software Testing," *Comm. ACM* **31**, 6 (June 1988), 687-691.

architecture and design of system's components and modules throughout development in order to prevent the product from exceeding the point of effective testing.

This need is especially pertinent to Web services, which are based on the use of inter-operable components. Web services can be commercially profitable only if their components are reliable both individually and in the various invocation combinations. Standard taxonomies (repositories of metadata definitions) and a standard ontology (metadata definitions with the associated business rules) are increasingly used to prevent an unwieldy expansion of component logic.

Another way to simplify a software component, or module, is to relegate a large portion of the logic to tables and other similar data structures. The result is leaner logic code, which is easier for developers and testers to understand and, thus, makes it easier to design an effective test set. The tables are generally devoid of procedural logic and can be directly mapped to the requirements or the design specifications during the developmental phases (construction, review, and testing).

For example, a means to test input data to ensure its validity can be embedded in the application logic or relegated to the environment—the database management system. If testing for validity is embedded in the software, then whether or not the application logic is correct is tested throughout the application. If the assertions about the value of data validity are checked in the metadata that drives the database management system, the validity can be tested more efficiently, more thoroughly, and more definitively, because the assertions are declarative rather than procedural—that is, they describe the attributes the data must have, rather than how they are to be used—and can therefore be mapped directly to the requirements specification and design documents produced in early phases of development.

The need to design for testability also surfaces later, in the maintenance phase. Every subsequent release of the software typically adds new requirements or modifies existing ones. When the changes are reflected in the application logic, they increase the complexity of the modules. Because of this increasing complexity, programmers lose the ability to comprehend how all the code works, and the changes will have unintended consequences. The programmers' inability to understand the module increases the difficulty of properly specifying tests for it, and, as a result, the reliability of the software suffers. The software becomes brittle and fragile, until the only recourse is replacement. Thus, software can be said to die of old age.

But in software designed in a table-driven style, the changes can be focused on the content of the table, which can closely track the revised requirements and design specifications. If modules are kept simple and small, adding new functions to them can be handled by adding new modules, rather than by increasing the complexity of a module. Although it is not possible for all applications to be designed with tables, such a design—that is, a simpler design—is a worthwhile objective.

Designing for testability can also be achieved by layering the functions. By grouping functions into logical layers, the layers are kept independent of one another, and each layer can then be tested more thoroughly because there are fewer combinations and permutations of inputs to test. When a particular layer needs to be modified, that layer can be tested more easily by itself (and a final integration test will ensure that the layers and modules still work together). This design principle parallels the procedure of internal and external auditors when creating systems of internal and external control. In such systems, auditing procedures are independent of the controlled process, so that the auditors can verify that the controls are in place and being properly used (**section 5.6**).

Designing for defensive programming means designing tests for conditions that should not arise but nevertheless often do. By designing these tests, failures can be anticipated, and the system can be made to fail "gracefully"—it will either reject the unanticipated input and cycle back to the next input, or it will terminate, with a useful error message. In this way, the developers can understand why a defect occurred and repair the faulty code or logic. Not all failures can be anticipated, and even the most defensively programmed systems encounter situations that "break" them (that is, result in a "nongraceful" termination).

The involvement of testing professionals from the beginning of the design cycle can ensure that the designs will incorporate the latest and best approaches for testability. Design is a human work product, and good design requires teams of well-trained, competent designers, whose qualities affect the quality of the software being developed. Are these people focused on business? Are they good communicators? Are they empowered? business savvy? technically savvy? Without the involvement of people with these qualities, any protocol or collection of methods is lifeless.[8]

3.3 Costs of Correcting Defects in the Software

The potential costs of poor quality software systems discussed in **Chapter Two** were affected by the costs and liabilities generated by malfunctions in the software. But defects in the software are costly also in terms of the effort and resources required to repair them. These costs are more straightforward to measure, because in most cases the activities of development (and maintenance) can be closely tracked.

A general rule of thumb is that the cost of correcting an error increases by an order of magnitude with each successive stage of the software development life cycle. If the cost of correcting a specification error is $100, then the cost of identifying and correcting the same error at the design stage will be $1000, and the cost of identifying and correcting it at the implement-

[8]Many other good design techniques exist that can improve quality, but they are beyond the scope of this report. The discussion here is limited to techniques that support effective testing and the correlation of requirements, design, and implementation.

ation stage will be $10,000, and in the final integration, $100,000. The cost of identifying errors after the system is in production needs to include both material costs (of correcting the error and distributing the correction to users) and the immaterial loss of reputation that has material effects.

In *Independent Verification and Validation*, Robert O. Lewis presented a post-hoc analysis of the cost of correcting defects in a real-time military application consisting of more than a million lines of code.[9] From that analysis, Lewis drew the conclusions shown in **Table 3-2**. Lewis also observed that that 62.5 percent of all the errors discovered during testing and integration were latent errors in requirement and design, and that 89 percent of the requirements faults were of one of the following types: inadequate information about operating rules; inadequate performance criteria; incompatible requirements; inadequate environmental information; inadequate system mission information. Within the design faults, at least 85 percent existed as one of ten types of processing or data errors.

Table 3-2

Lewis's Conclusions

Phases of Software Development Life Cycle	Average Cost of Error Correction
Requirements	$200
Design	$500
Testing of coding and unit	$977
Integration and testing	$7136*

Robert. O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992), 178-279.

*This amount represents the 25 percent uncertainty bounds.

These data tell an unambiguous and not too surprising story: that identifying and correcting errors during the stage of the software system's development life cycle where they are generated can substantially reduce the overall costs of development. According to Lewis, "If the developer…had concentrated on the 15 out of 76 error types that caused 85% of all latent requirement and design errors discovered during integration, this effort could have saved over 50% of the total cost of integration changes…or about 20% of the total cost of integration…[or] 9% to 11% of the total cost of the program."[10] The elimination of considerable effort in the later phases of development does more than save dollars; it allows compression of time frames, which produces the business benefits of accelerating the deployment of the system. Thus, the business

[9]Robert O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992).

[10]R. O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992), 278-279.

case for appropriate testing early in the development life cycle translates into lower development costs and greater business value.

3.4 Agile Programming

Several emerging software development methodologies, generally characterized as either lightweight or adaptive, have as their conceptual basis the recognition of the difficulty that many organizations have in implementing full-scale, “heavyweight” methodologies (which are rigid enough to require unnecesessary activity) and the recognition of the cost issues discussed in **section 3.3**. Proponents of these methods point to the unpredictability of the process of software development, beginning with the difficulty organizations have generating correct, complete, and stable system requirements. If, despite efforts to test the requirements early in the development life cycle, the requirements change, the development process will inevitably be longer and more costly than projected.

Among the new approaches that have emerged are agile development and extreme programming (XP). These approaches divide projects into very small pieces, each of which can be fully developed and tested in weeks (or, at most, a few months). The premise is that successive stages of development will produce the system that both users and a business actually need.

In a direct response to the cost issues described in **section 3.3**, these approaches include testing before coding as the only cost-effective way to deal with the dynamic nature of distributed computing, in particular, in the Web services environment. The tests as well as the resultant code are to be developed iteratively, so that when the code has been completed, the test is complete as well and in each iteration the code always needs to pass the test.

To date (mid-2003), these new approaches have been shown to work well when used by relatively small development teams (of fewer than 40 developers) and in rapidly changing business situations. Whether they can be applied in large environments remains to be seen.

3.5 FDA-Mandated Software Validation

A validation program is intended to produce documented evidence that will provide assurance that a specific process will consistently result in a product that meets predetermined specifications and has certain quality attributes.

When a system is to be used, say, in the manufacture of a pharmaceutical product, the definition of validation necessarily includes the proper interpretation and application of the FDA’s Good Manufacturing Practices, so that the manufacturing facility will consistently meet or exceed the practices required. Validation applies to systems, processes, and programs that support the manufacturing process.

In Regulation 21CFR Part 11,[11] the FDA established requirements for the use of electronic records and electronic signatures in systems that are subject to validation, to establish the accuracy and authenticity of the records and signatures and provide for long-term preservation of records. Because the FDA's validation rules describe how the stages of a design life cycle need to be documented, they address more than the testing stage. For example, during a validation inspection, specification and design documents are examined to demonstrate that a reasonable process of software development has been followed. Test plans and test results also are examined.

But inspections can be pro forma. The assumption of the rules seems to be that following a reasonable development process will "provide a high degree of assurance that a specific process will consistently produce a product meeting its predetermined specifications and quality attributes." It is possible, however, to follow a process formally without achieving a "high degree of assurance" of its ultimate performance.

The system Lewis examined (**Table 3-2**) was a military system, not a pharmaceutical or medical-device system. But the program involved in its development used a formal software development life cycle, followed formal quality procedures, and was examined by an independent verification and validation group. There is no question that the work products of the various developmental phases would have complied with the FDA's validation requirements; had the agency's inspectors examined the system, they undoubtedly would have passed it. Yet more than half the errors Lewis found in the integration and testing phases were latent requirements and design defects. There was no actual assurance that all the system's defects were discovered and corrected—only the system's eventual failure on the battlefield would have exposed these latent defects. Thus, questions arise: Is this a validated system? Does it have "a high degree of assurance that its development process did produce a product meeting its predetermined specifications and quality attributes"?

Validation is similar to software quality assurance as an internally focused discipline that comprises a "set of methods and procedures that enforce internal rigor in the software development process." Internal rigor is commendable and a step in the right direction. It may even help an enterprise to pass inspection according to the FDA's requirements for validation. But internal rigor will not guarantee that the resultant system works correctly all the time (there are no such guarantees). Companies and individuals who rely on the particular system for medical or commercial purposes remain at risk should it fail.

From the perspective of the FDA, the final defense of validated systems is that they need to retain an operational history that tracks and records all operational anomalies. This process should eventually discover systems that are not operating correctly, and the faults should then be corrected. Yet a system can be put into operation although even after validation by the FDA's requirements it is not operating completely correctly. A well-designed program of testing that

[11]Part 11 of Title 21 of the Code of Federal 19 Regulations; Electronic Records; Electronic Signatures.

tests all phases of the software development life cycle for the quality of the work products can generate a very high degree of assurance that the system will meet its predetermined specifications and quality attributes. And such a program can ensure that the system will comply with the FDA's validation requirements.

The crucial point is that the testing program cannot be pro forma. It needs to address the content being developed in each phase. Its requirements need to be examined in depth, for consistency and completeness of the functional objectives, operating rules, performance criteria, and environmental criteria. The design needs to be tested against the requirements as well as for adherence to good design principles. Unit testing and code review need to be conducted by experts who are familiar with the requirements and design and are qualified by experience to identify actual or potential problems with the implementation chosen by the developers.

Many pharmaceutical companies assume that the cost of developing a validated system is double that of developing a nonvalidated one. The cost-doubling is unnecessary and is the result of an ill-conceived quality philosophy on the part of the companies. The costs of a full testing program can be recouped by the dramatic lowering of the need for—and therefore the cost of—reworking in the later phases of the software development life cycle. If the quality management program and the testing program are proper and comprehensive, meeting the specific requirements of the FDA will not add much to the final cost.

The FDA imposes requirements that go beyond the specific needs of high-quality software. It requires the "demonstration of quality broadly and publicly."[12] This is a reasonable expression of government responsibility for the health and welfare of its citizens, but one that adds to the cost of the validated system. The FDA requires, for example, that the test documentation include such details as the recording of all test values (not just an indicator of pass or fail), which imposes an incremental work load. But the bulk of the FDA's validation process involves documenting that the software development process is well controlled, which is what a proper program of quality management is all about.

[12]Personal communication.

Chapter Four

Four Principles of Software Quality Management

The description here of four fundamental principles of software quality management that software development organizations should follow to produce higher-quality software-system products does not put forward a specific, quality-related program that a company might adopt. High-quality software can be developed without the use of a formal laboratory or clinical program, a total quality management program, or a six sigma vision. But the value that can be obtained by following the general thrust of these programs is indisputable. Software development is a business process that needs to be subjected to continuous improvement. It needs to be both a repeatable process and one with effects that management can measure.

A consequence of the view that software development is a business process is that quality needs to be viewed as a business requirement, with well-defined cost and risk targets. Although nearly unlimited resources can be devoted to a quality management system, for most (maybe all) organizations, this is neither sensible nor practical. Instead, quality can be regarded as a risk management initiative, with a twofold goal: first, the risk of system defects needs to be lowered below a predetermined threshold; and, second, additional resources should be expended only if the business value of the higher quality of the software exceeds extra quality costs. On the one hand, the quality of mission-critical systems, the failure of which could endanger human lives, should be the highest possible. Videogame software, on the other hand, might be permitted to fail. The issue is whether the rate of failure would lead to a competitive disadvantage and the loss of revenue or profit.

The discussion of the types of metrics used to measure the quality of software opens with the description of metrics appropriate to each phase in the software development life cycle and of methods for setting targets and proceeds to a discussion of the overall process of quality management, the need for trained professionals for assessing quality and for testing, and then to such organizational issues as the independence of quality and testing organizations.

Related to this discussion is the need for active management of the process of development by both the business executives and the IT executives. There are checkpoints, and there is a series of decisions to be made at the conclusion of each phase of development. An oversight committee needs to be established, which has the responsibility and is accountable for these decisions, and the rank of its members needs to be commensurate with the business benefits, costs, and risks of the system under development.[1] Management, from the executive level down, needs to reject the

---

[1]Further discussion of IT governance is beyond the scope of this report, but the reader is referred to another work by the author, *A Holistic Framework for IT Governance* (Cambridge, Mass.: Harvard University Program on Information Resources Policy (February 2000), [On-line]. URL: http://www.pirp.harvard.edu/pubs_pdf/popper-p00-1-pdf.

notion that because software faults are inevitable in any complex system, producing defective software is acceptable.

The discussion of the thorough testing required in all phases of the software development life cycle is very important, but these elements have been noted elsewhere.[2] What is new is the way information about and the discussion of the management process, the need for professionals to assess quality and perform testing, and the need for independence are brought together in the discussion here. The discussion begins with the need to define quality metrics and the associated targets for each stage of the software development life cycle.

4.1 Principle 1: Measure Quality during Each Phase of Development

Requirements Specification. The ten criteria and metrics for evaluation of the quality of a software system for the requirements specification phase of the software development life cycle are listed in **Table 4-1**. Their purpose is to ensure that the resultant system will satisfy the expectations of the end-users. Although many of the criteria cannot be determined algorithmically, a three-point scoring system can be devised as a way for the users, developers, and independent testers (internal and external) to assign a score to each criterion. If the quality of the product of the requirements specification phase is only "average," then the resultant system is unlikely to achieve a higher score and, indeed, is liable to contain serious defects. The business and the IT management need to establish a checkpoint that will prevent a project that does not meet the standard of the quality phase from proceeding to the next phase, which is the design phase.

The objective of the metrics and criteria is to ensure that requirements are correct and can be implemented as planned and to identify requirements that need to be modified, added, or deleted in order to attain a suitable requirements specification. The targets need to reflect the objectives. A simple scoring system (with scores ranging from A to C) can be used. The standard for passing might be the following: all grades need to be at least B; a C forces reworking to fix the problem. Of ten possible grades—one for each of the ten criteria or metric—five need to be A (but if a weighted average is used, the total score needs to be at least A−).

Design. In the design phase, critical decisions are made concerning the organization of the system, the standards and tools to be used, and the way the system is to function. The criteria and metrics for evaluation of the quality of the design need to address the key elements listed in **Table 4-2**. The objectives of the metrics and evaluation criteria for design are to ensure that the design will comply with the requirements—that is, that it is consistent, complete, correct, and can be

[2]See, for example, Robert. O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992), 282; and, with reference to the ISO, see the Praxiom Co. Web site, at URL: http://www.praxiom.com/, and URL: http://www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.

Table 4-1

Ten Criteria and Metrics for Evaluating the Requirements of a Software System

1. Are the requireents clearly and completely stated? Are the terms well defined, precise, and consistent? Are the definitions based on a standard ontology or taxonomy or, at least, a fully documented ontology or taxonomy?
2. Are the requirements complete? Is a use-case analysis included?
3. Do the requirements adhere to all appropriate standards?
4. Do they incorporate external specifications, e.g., global requirements of the enterprise?
5. Have they been confirmed by means of quantitative analysis, modeling, simulation, rapid prototyping, or benchmarking, among other means?
6. Do they reflect the social and cultural context of the intended user community, so that the system is likely to be integrated into their business practices?*
7. How easy will it be to construct tests to validate that the requirements are being met? What tests will be run?
8. How clear are the secondary objectives, e.g., logistics support, personnel and their training, delivery requirements, among others?
9. Are there alternative system definitions that might deliver more efficient or more effective solutions?
10. What are the primary risks? What risk mitigation strategies are included? Are the risks manageable?

*See "User Centered Design,"[On-line]. URL: http://www.rmit.edu.au/bus/rdu/supriya.

implemented; and to identify design elements that need to be modified, added, or deleted in order to attain a suitable design specification. The targets need to reflect these objectives. A scoring system can be used that is similar to the one used for the requirements specification phase. In the design phase, however, adopting a numerical score is appropriate, because the quality of design represents a continuum, whereas requirements are better scored as either pass or fail. In the design phase, as in the requirements phase, a checkpoint needs to be set at the end of the phase in order to prevent a project with a quality of design that is lower than the target from being carried into the implementation phase.

Implementation. The metrics to be used for evaluation of the quality of the implementation phase (**Table 4-3**) need to address the system code and the methods used to produce it, as well as the documentation. Good grades for these aspects of quality ensure that the software system will operate as planned and that it can be maintained during its life cycle.

The objectives of these metrics and evaluation criteria are to ensure that the software will comply with the design, that it is consistent, complete, correct, resource-efficient, well organized, and well documented, and to identify code elements that need to be modified, added, or deleted in order to attain a suitable implementation. The targets need to reflect these objectives. A scoring

Table 4-2

Criteria and Metrics for Evaluating the Design of a Software System

Can all the requirements be traced in the design? Can all aspects of the design be traced to some requirement? This is the time to create a traceability matrix that will initially relate requirements to design elements and eventually should relate them both to elements of implementation and integration.
Are the system interfaces correct and complete?
Are the data flows correct and complete?
Are the control flows correct and complete?
Are the algorithms correct and complete?
Is the database design correct and complete?
Is the user interface correct, complete, convenient to use, and easy to learn?
Are the resource requirements (processing speed, memory and disk allocations, etc.) within acceptable limits?
Were other design alternatives considered? What criteria were used to select the winning design?
Does the software development plan conform to relevant standards? Is it appropriate for the design?
Risk assessment: What are the primary risks? What are the risk mitigation strategies? Are the risks manageable?

Table 4-3

Metrics for the Quality of the Implementation Phase

Is the code consistent with the design in relation to both the metadata and algorithms?
Does the code conform to specified standards?
Were all the specified tools used and used appropriately?*
Is the logical structure correct?
Are the data elements and names used in the code consistent with the data dictionary?
Are the sample inputs and outputs used by the programmers in their unit testing appropriate, complete, and correct?
Do the coded algorithms conform to the design specification?
Are the resource requirements of the functioning code consistent with the design projections?
Were the correct versions of all system tools (compilers, databases, operating systems, etc.) used?
Is the software library well designed and controlled? Were appropriate procedures followed for release and version control?
Were all necessary documents produced? Are they properly archived? Are they readable and understandable? Have they been updated as the software has been updated?

*Static analysis tools can help in this regard.

system can be used that is similar to the one used for the design phase, with one major difference: if a defect is discovered during the design phase, it needs to be repaired at the point of discovery. But isolated defects—those with effects that do not extend beyond the specific module—that are discovered during implementation may be corrected later, in the integration phase, where it is likely that others also will be found. This approach reduces the overall cost of eliminating defects.

The management checkpoint at the end of the implementation phase is not an absolute "go-or-no-go" point, but it is critical to the management of known faults. A clear, shared control document (or database) is needed that lists exceptions, work plans to resolve them, target dates for resolution, and progress in accomplishing goals.

Integration. The integration phrase is focused on the actual software and how it performs, rather than on methods, tools, libraries, and so on. Yet at this point, the methods and tools need to be under control and libraries need to be updated, even though the focus is on is whether the system executes the desired commands and functions in the desired way (**Table 4-4**).

Table 4-4

Criteria for Evaluating the Quality of Integration

Does the system meet all the functional requirements, all the performance requirements, all the environmental requirements?
Is the user interface suitable? Can users use the system efficiently? Can they learn it easily?
Does the system fail gracefully—that is, does it respond in a reasonable way when presented with invalid inputs or instructions? Are data lost if the computer is suddenly stopped? After a failure, how difficult is it to restore the databases and the system's inputs?

In the integration phase, the complete subsystems (systems integrated from component modules) and the entire system are tested. The test plans need to be designed and constructed in advance; to ensure that the translations of requirements into design and, then, into software were correctly performed, the tests should be based on the original requirements specification. If a new system is replacing an old one, then other parallel operations need to be performed, to compare results obtained by the new system to those obtained by the old one.

The traceability matrix, a tool that correlates specific requirements with associated design elements, code elements, and test cases, needs to be developed incrementally during the phases of the software development life cycle. It is an important asset for use by the development team, because it enables the repair of defects during the integration testing with only a small risk of introducing new ones into other, related portions of the system.

The evaluation of the results of the integration tests is more complicated than the evaluation of those of previous phases. The question at this point is whether or not the system is ready for

production. Both the functional and the performance requirements need to be categorized as essential—that is, if these requirements are not met, the system cannot be used—or as optional—that is, the system can be used even if some modification may be needed for it to be judged fully ready for production.

These judgments are often difficult to make. A system that is almost functional but imposes a considerable burden on its intended users may result in a degraded business performance, dissatisfied customers, and lost revenue for the developer's business—and the losses can be permanent. Whether to put the system into operation is a decision to be made by the business; the software development and testing teams should be very clear in their presentation to the executives, to enable them to make as informed a decision as possible. **Table 4-5** presents a possible structure to use to present the benefits, costs, and risks associated with a system. The executive decision of whether or not to proceed (go or no-go) may depend on a few key factors, so the presentation of the factors needs to be carefully organized. The factors need to be carefully and clearly identified in the presentation, which needs to be focused on them and, secondarily, on the activities that are needed to support every potential decision. In the integration phase, as in the implementation phase, a tracking mechanism needs to be established to track known exceptions, so that the system will conform to the specified requirements in a reasonable time.

Table 4-5

Possible Structure for Presentation of Benefits, Costs, and Risks of the System

Which new functions work? What is the business benefit of these functions?
If there is a delay in putting the system into operation (e.g., because regulatory requirements or contractual agreements have not been met), what are consequences for the developer and for the user?
What does not yet work? How can the business circumvent these missing functions?
How efficient will the users be? Do staffing levels need to be adjusted for the use of an interim system? What are the projected costs?
How efficient will the system be? Are the customer's hardware resources adequate to use the new system? Should additional resources be obtained for use of the interim system?
What is the risk of failure of the new system? If failure should occur, what contingency plans are in place?

The list in the table indicates the evaluation metrics to be applied to a system as its development proceeds through its life cycle. The organization developing the system needs to build a database of these metrics for all its projects, in order to be able to identify needs and opportunities for improving its processes. The choices listed in the table need to be applied consistently (for which reason they will be scored) and across the developer's organization. In some instances, the metrics and scoring may seem burdensome, even inappropriately so, but the value of creating good metrics that will apply organizationwide is high, and all future projects

will benefit from their creation. This aspect of the total quality management program is a major theme of the principle 2 (**section 4.2**).

This discussion is focused on the initial development of a software system, but almost all systems undergo subsequent development as changes in a business and in technology create the need for new releases of the software. Managing changes to the system is itself an important organizational skill, as various phases of the software development life cycle need to be revisited. If the initial development was well managed and appropriate documentation was created, then the subsequent development cycles will benefit from the investment in them. This benefit accelerates as the system goes through production, because without the rigorous development and attention to quality, the system would become unstable.

The development of a useful, coherent program of measurement to be applied as testing to the full software development life cycle can derive substantial benefit from guidance by professionals who can focus exclusively on the evaluation and testing of software. Their experience across a wide variety of industries and development organizations enables them to construct a rigorous and practical program for evaluation.

4.2 Principle 2: Establish a Thorough Quality Management Methodology

The establishment a thorough method of quality management is essential to the process of continuous improvement by the development organization.

A development organization that is focused on quality seeks improvements to its processes and procedures (**Table 4-6**). Software defects are analyzed to uncover root causes, and the development process is constantly adjusted to reflect lessons learned. This is particularly true for recurring defects; changes in development practices and organizational behavior are necessary to eliminate or, at least, control root causes of defects. Defects can occur in all phases of development, from requirements specification on. The root-cause analysis applies across to all the phases and processes of development and needs to be constantly improved.

The meticulous documentation of processes and activities is a key need. To institute procedures to prevent a repeated appearance of defects, the developer needs to determine whether certain negative effects were the result of processes of development (**Table 2-1** and **section 2.3.8**).

Clearly, the development of quality software by an organization focused on the quality of its product is not simple work. It requires both a commitment to excellence and persistence, a steady grasp of the details. The root cause of each defect in each part of the software system needs to be discovered and removed. The resultant analysis needs to lead to an adjustment of work habits and development processes, to ensure both the development of quality software and that an organization maintains its focus on continually improving both itself and the degree of quality of its

Table 4-6

Procedures to Prevent Recurrences of Defects

Discover how and why defect was inserted into the product, and
Consider how adjustment of the process can prevent recurrences:
To understand how a defect was inserted into the product requires thorough documentation of development process, activity, and defects, as well as analysis of the root cause of the defect.*
Implement quality methods that will define procedures used in software development and testing to ensure production of highly reliable products; design factors to be addressed by the method used include the following life-cycle processes:
Configuration management
Document management
Standards requirements
Certification requirements
Metrics
Testing design
Independent verification, validation and testing
Root cause determination
Process improvement strategies

*J. Dennis Lawrence and Warren L. Persons [preparers], "Survey of Industry Methods for Producing Highly Reliable Software," U.S. Nuclear Regulatory Commission, Fission, and Energy Systems Safety Program, Lawrence Livermore National Laboratory (Pub. No. NUREG CR-6278UCRL-ID-117524, Aug. 29, 1994), 2-5, 22-24.

products. Oversight committees, as discussed at the start of this chapter, should have responsibility for these issues on a cross-project basis; semiannual review of quality processes and results is appropriate. The IT executives and business executives need to be held accountable for the organization's overall quality metrics.

Is such rigorous management worth it? Does a formal methodology impose its own overhead and costs? The simple answer, given in section **3.3**, is that the benefits of early fault detection and correction can exceed the costs of the quality program that finds and corrects these errors. And, as discussed in **section 2.2**, the cost of defective software is immense. Investing in techniques to reduce the rate of defects should be very effective. As to whether and to what extent the techniques discussed here will accomplish the objective of lowering the rate of defects, there is a growing body of evidence that they will (for anecdotal evidence, see **section 5.7**).[3]

The methods needed to support a communication platform that will be shared by the technology provider, the users, and the testers are a key element in ensuring the development of

[3]URL: http://www.sei.cmu.edu/tsp/; also Watts Humphrey, "Pathways to Process Maturity: The Personal Software Process and Team Software Process," [On-line]. URL: http://interactive.sei.cmu.edu/Features/1999/June/Background/Background.jun99.htm.

quality systems. One source of failure in the processes of software development is lack of communication among the various stakeholders in a system being developed. Insufficient communication can lead to poor requirements gathering and inadequate documentation.

A thorough, quality-based development method requires clear communication from the start of project through its deployment and maintenance. Effective communication with customers, as well as among team project members, will ensure coordination of the requirements and expectations among the stakeholders. A good communication platform is an important part of the successful development and deployment of software systems of high quality. The communication platform should be based on the documentation produced in the software development life cycle—documents such as the requirements specifications, design documentation, and test plans—which creates an environment and a method for the stakeholders to use to communicate and define an integrated view of how the product should look and what form its processes should take. This documentation can function as a centralized repository to be referred to at later points in the development processes. It will allow searching (tracing) within the phases of development and thereby ensure the direct effect of the phases of planning and execution of the development cycle.

For example, communication between users and developers is critical to quality, and there is a natural tension between the precision of a specification (which can be improved by employing a more mathematical notation) and the ability of users to understand and properly approve or modify the specification. A possible approach to solving this is to construct the overall test plan and the test cases for the system, based on the requirements specification. The tests are rigorous, from the developers' perspective and yet understandable by the users. User approval of the test plan should be a prerequisite for a "passing grade" for this criterion.

One important aspect of the communication platform is that it leads to the establishment of common metadata and the use of data dictionaries throughout both the system and the enterprise. The data are at the core of all systems, and inconsistency within the data is a major cause of defects in software. The use of common metadata can prevent many problems. The metadata repository, which should be established at the time that the project is initiated, can evolve through the phases of requirements, design, implementation, and testing.

4.3 Principle 3: Employ Professional Testers with Experience in Testing for Quality

Experience, expertise, and training in the planning and deployment of testing procedures that are focused on the quality of software are essential to the development of software of high quality. The discipline of testing is not obvious, as the poor quality of most systems available

today has shown. Testers—professionals who master this discipline—need to know what to measure, how to measure it, and how to assess the root causes of poor quality.[4]

The processes that testers are responsible for carrying out are complex and require a multi-disciplinary background, education, and set of skills (**Table 4-7**). They also require an ability to manage and execute them. Testers need to have full knowledge of new technologies and platforms, available test tools, relevant standards and protocols, and of the legal implications of quality based on the development organization's industry.[5] Such expertise is critical and encompasses knowledge of industry-specific practices, standards, and behavior necessary for real-world testing as well as an understanding of the legal implications, regulatory requirements, and the customer's expectations unique to the particular industry.

Successful planning and successful execution of tests require the software testers to consider the software and its functions, the inputs and their various combinations, and the environment in which the software will be used. This evaluation requires the testers to possess a high degree of technical sophistication. Skillful testing requires certain kinds of knowledge and skill sets (**Table 4-8**).

The need for professional testing expertise is particularly important for the maintenance of the software. Most, if not all, applications undergo change when they are in operation, which is when new business requirements and user requirements can arise. Each release of the software needs to be tested to ensure that all the new functionality works correctly and that the functions previously in place continue to work as before. Software development companies routinely apply regression testing to new releases: the set of regression tests comprises all tests of previous functionality that the new software release has to pass. (Regression testing is important also in each phase of the development cycle, to ensure that the correction of defects does not generate new defects elsewhere within the system.) A number of software testing tools are available to assist in the automation of the testing, in particular, regression testing. Because acquiring and implementing the appropriate testing tools requires skill and experience, these needs provide another reason for developers to bring into the process professional testers, either in-house or outsourced.

---

[4]"The Cost of Poor Quality" *Computer Finance* 9.06 (January 1998), [On-line]. URL: http://www.ispw.com/articles/Computer Finance.htm.

[5]James Bach, "Outsourced Vendor Analysis," Satisfice Consulting (2003).

Table 4-7

Processes for Which Testers Are Responsible

Test planning and execution
Defect and quality measurements
Predictive quality estimates
Software reliability predictions
Statistical analysis and reporting

Table 4-8

Testers' Knowledge and Skills

Knowledge of programming languages
Knowledge of platform technologies
Knowledge of algorithms
Development skills (e.g., coding)
Quality assurance skills, such as attention to detail, patience, and organizational skills
Communication and collaboration skills
Leadership skills
Experience in project management

Professional testers not only detect defects; they also aid in the creation of quality processes within an organization. The tester or testing organization needs to use measurement and statistical analysis to build a knowledge base that will enable the organization to apply continuous process improvements. They need to be able to measure the volume of defects, their severity, and their origin. To do so, testers need to have a deep knowledge of programming languages, algorithms, and configurations. For true quality assurance, professional testers need to be able to understand the underlying causes of defects and to define organizational and behavioral changes that can promote the attainment of quality.

Testers or a testing organization are important to knowledge dissemination; they can encourage a companywide acceptance of new development processes that are focused on quality and of management techniques to use throughout the development life cycle. Experience, expertise, and authoritative knowledge are crucial to the relationship of testers to the development organization. Testers need to possess and confidently display strong technical knowledge and skills in order to sustain their point of view, which may, at times, oppose that in the organization. By their professionalism and expertise, the testers can elicit the respect necessary to their success in improving both testing and processes.

Expert professional testers will have knowledge of developments within the customer's industry and will participate and be leaders in quality-assurance industry conferences and professional forums, especially testers in regulated industries that have very specific quality and testing requirements. The software industry is constantly changing as new technologies appear and regulations influence the testing requirements. Such change means that a dedicated organization will be one that specializes in software development and has the knowledge and experience to conduct research and to assess and integrate the appropriate best practices and testing methods into their repertoire.

4.4 Principle 4: Ensure Independent Quality Management and Testing

Whether or not a system meets its requirements should be determined by independent testers. Neither the developers, who are being paid to produce a system, nor the technology vendors, who are selling off-the-shelf products, can have the objectivity necessary for an impartial assessment.

Critical to an assurance of the quality of software is the independence of those assuring it. Developers require the introduction of a new voice to represent the users and address their the needs and desires. When either developers or technology vendors are the ones who determine whether or not a system meets its requirements, the examination and analysis may lack the clarity of vision that inheres in independence. The intended users of a system ought to be able to have a direct role in the development process, and as the buyers they should have final say on whether the delivered system is acceptable. But even a determination that the requirements are correct and that they accurately reflect the intent of the users requires independence—ideally, independence from both developers and users. An objective and, as far as possible, unbiased view can promote a good balance among users, vendors, and technology developers.

There are several options for achieving independence. One is to establish a separate quality assurance organization that will report directly to the senior IT executive, as the internal audit function generally reports directly to the chief financial officer. Internal audit, however, also has a direct relation to the board of directors; by analogy, quality assurance should have direct access to the chief executive officer or other senior business executives. A hybrid solution might be to have a portion of the testing conducted by the development company, with quality assurance certifying its test planning and execution, and to have external testers audit the internal testing process. This procedure would instill a minimal external independence.

Most development companies are aware of their need to have an independent point of view in the testing of their software. The addition of such independence may be accomplished within the company by attempting to ensure that the requirements analysts, software developers, and testers are organizationally independent of one another. Although this situation would be preferable to interdependence, the addition of a truly independent group would be ideal and would ensure a balance between the interests of the users and developers. External review of software code and development processes can lend insight that may otherwise not be available from within the software development company. An external review is especially important during integration testing (**section 4.1**), when critical go-or-no-go decisions must be made.

Such an independent viewpoint is even more important to organizations that are having difficulty establishing adherence to their chosen software development method and to rigorous quality practices. An internal quality-assurance group usually has only a limited effect, whereas an external group usually can have greater leeway and the capacity to force compliance.

Having independent professional testers develop the final acceptance tests during the requirements phase allows the development company to amplify the benefit of objectivity. It allows users to see how their requirements are translated into tests and provides confirmation of the quality of the requirements. By initiating the activity of the independent testers during this phase, users can request important changes early in the development process, when the cost of adjustments is minimal. Thus, an iterative process is created that opens lines of communication between developers and users. The testers can serve as liaison or interpreter, explaining to the users how the developers view the requirements the users have communicated and allowing the developers to understand how the requirements documentation that they created will conform to the functionality and use of the system. Iterative adjustment of requirements and specifications result in a greater likelihood that the end product will be of high quality.

Chapter Five

The Role of Independent Testers

In many (perhaps most) cases, it is nearly impossible for a development company to make such decisions objectively. Deadlines, management pressure, and the customer's needs can directly affect the developer's decisionmaking processes. This is particularly important in the pharmaceutical and medical-device industries, where errors in the software can lead to loss of life.

Because the requirements phase of the software development life cycle can be broken up into four parts, it is useful to understand which part or parts independent testers can add value to in ensuring the quality of the end product. The four parts are the following:

1. **Conception:** To identify needs, examine architectural possibilities, and develop the outlines of the solution to be developed;

2. **Generation:** To define software requirements and produce the requirements specifications;

3. **Analysis:** To determine the potential resource requirements, the effect on the technical environment , and assess the implementability, testability, and acceptability; and

4. **Approval:** To evaluate risks and benefits, decide on resource expenditures, and establish baseline targets for costs and time-lines.

In the stages of analysis and approval, independent testers can provide both the developers and their customers with an objective assessment of the proposed requirements and how they will affect the customer's operations by defining the effects on resources and on the readiness of the requirements, as well as by pointing out potential errors and challenges.[1] An important part of the testing plan is clarification of the performance standards that are in place for the determination of quality. Possibilities include standard-less review, standard-less independent audit, and conformity with industry standards, regulatory standards, or client-specified standards.

For work in the requirements phase to be successful, professionals who are expert in assessing the quality of software need to focus on preventing the introduction of defects into the development process, rather than on trying to "test" quality into the software after code has been written. Beyond simple software testing, validation of the software needs to establish confidence that the software is fit for its intended use. These activities are usually part of the testing and quality assurance within a development company. Although for many developers this function is carried out in-house, through peer review, team review, or even departmental review, it is nearly

---

[1]J. Dennis Lawrence and Warren L. Persons [preparers], "Survey of Industry Methods for Producing Highly Reliable Software," U.S. Nuclear Regulatory Commission, Fission and Energy Systems Safety Program, Lawrence Livermore National Laboratory (Pub. No. NUREG CR-6278UCRL-ID-117524, Aug. 29, 1994), 22-24.

impossible for the developer to review the quality of its own specifications or code objectively. Self-criticism is an issue when user dissatisfaction arises. The FDA's principles for software validation clearly state that "self-validation is extremely difficult. When possible, an independent evaluation is always better, especially for high-risk applications."[2]

Independent testing is important, beginning with development of the test and ending with the assessment of the results. For the results of the testing to be valid, objective pass–fail decisions are needed. In an independent audit of test results, the testing organization is determining the kinds of tests to run as well as what passing or failing is. Independent tests can be responsible for documenting all test procedures, data, and results to be used for review and for objective decisionmaking after the testing. They can provide defect analyses and suggestions for improvements to the development process. They can have a role in determining whether the product is suitable for commercial distribution. In all, independent professional testers have four important roles in the software development life cycle: to supply experience, skill, expertise, and industry awareness; to analyze organization's needs for testing; to implement comprehensive testing method and train others in its use; and to develop and execute test programs.

5.1 Supply Experience, Skill, Expertise, and Industry Awareness

Software testing requires unique skills and experience. Testers need full knowledge of the implications of the testability of new technologies and platforms. They need to know about available test tools, relevant standards, and protocols. They need to know the legal implications of quality within the developer's industry.[3] A professional organization of testers focused on software testing is ideally suited to this work. The organization not only would be aware of leading-edge testing technologies and relevant standards but also would be likely to be active in the quality-assurance community, be participants in industry conferences and professional forums, and be a force in industry developments. Its position would allow it to empower client organizations to use outsourced testing services as a tool for building competitive advantage into their own testing methods and organizational processes.

5.2 Analyze Organization's Needs for Testing

Outsourced testing has multiple meanings. Most simply, outsourcing means delegating work to one or more persons outside of an organization's immediate control. This means a firm or another in-house group that has organizational independence. Development companies can outsource all testing, testing for one project, or one testing task within a project. Outsourcing may remove work from within the core organization, supplement the organization's resources or

---

[2]Dept. of Health and Human Services and the FDA, "General Principles of Software Validation; Final Guidance for Industry and FDA Staff," (Washington, D.C.: Dept. of Health and Human Services and the FDA, U.S. Gov't Printing Office, Jan. 11, 2002), 10-12, [On-line]. URL: http://www.fda.gov/cdrh/comp/guidance/938.html.

[3]James Bach, "Satisfice Test Model Strategy," [On-line]. URL: http://www.satisfice.com/tools/satisfice-tsm-4p.pdf.

expertise, or both, in a particular area, or import an objective point of view into the development process. Outsourcing may also be used to satisfy internal requirements of an organization or contractual obligations for independent testing or to fulfill the requirements of a regulatory or certifying agency.

5.3 Implement Comprehensive Testing Method and Train Others in Its Use

There are four phases involved in building a test plan. In the first phase, the tester needs to simulate the interactions between the software and its environment in order to identify and test the various interfaces—human, software, file-systems, and communication interfaces—and the corresponding inputs that will interact at each interface.

In phase two, the tester selects test scenarios. Testers strive for coverage of code statements and inputs, executing each line of code at least once while applying each externally generated event. However, if code and input coverage alone were sufficient, there would be very few defects. Testers are concerned with execution paths, not simply with individual code statements, and with input sequences, as opposed to simply input domains. The challenge testers face is that there is an infinite number of both execution paths and input sequences, too numerous to for each one to be tested. The testers therefore need to determine the adequacy of criteria for the test data to ascertain the best possible set of test data set to represent such infinite possibilities. To accomplish this, testers generally seek a set that will detect the greatest number of defects, especially the most serious defects (from the perspective of risk or financial liability). They may also evaluate typical use-case scenarios—those most likely to occur often in real use. In this way, testers and testing attempt to ensure that the software works as specified and that the most serious defects and those most likely to occur frequently have been detected and removed.

Phase three comprises two parts, running test scenarios and evaluating them. Testers first convert the identified tests into executable form in order to simulate user action. Often, the test scenarios are automated to cut down on labor-intensive, error prone acts of manual application. Second, in scenario evaluation, testers compare the actual output of the automated testing with the expected output as delineated in the product specification. The comparison may be automated in whole or part; in some complex situations, manual intervention is required.

In phase four, the progress of the testing is measured. Although testers tend to track their progress by counting items such as the lines of code tested and the number of defects found, such numbers may yield little insight into the progress of the testing process. To track progress, testers may seek to examine structural and functional completeness or the adequacy of test-data criteria for each area tested. Measures might include the degree of code exercised, the internal data initialized and used, and the use of common scenarios and application of inputs. These measures, however, do not provide the testers with a quantitative understanding of the number of defects left within a system and the probability that the defects will be discovered in the field. Such a measure is vital for testers to know when to stop testing. Of use in this quantitative analysis are

the issues of testability—the ease of testing and locating defects—and reliability—the predictability of patterns of future failure based on previously obtained data. These two measures enable testers to gauge the progress of testing of both structural and functional areas.[4]

Once the test methodology has been established, it is the job of the testing organization to communicate it to the company—that is, to educate the developers about the testing processes as and the role testing has in monitoring development at every phase of the life cycle. It is very important that developers understand how they can simplify their jobs and improve the quality of their product actively working with the testing organization, especially at the point of requirements analysis.

5.4 Develop and Execute Test Programs

In some organizations, the role of a third-party tester is quite hands-on. The testers will not only have an active role in the development and planning of the test methods, but they also will actually develop and execute the specific tests. This might be the case in a development project that is particularly sensitive or mission-critical.

5.5 Internal or Outsourced Testing? When, Where, and Why

Many factors enter into the choice for a particular project of whether to use outsourced or in-house testing (**Table 5-1**). These factors can be evaluated on a ten-point scale and then combined into a single means to measure the need for outsourcing (**Table 5-2**). Factors may also be combined according to various decision rules. If, for example, 6 of 10 factors each have a score of 5 or higher, then the use of independent testers is warranted; if, however, 4 of 10 factors each have a score of 8 or higher, the use of independent testers is warranted; if the weighted average is 7 or higher, then the use of independent testers is warranted. These rules are merely suggestive; each organization should develop its own suitable risk-assessment procedure.

The conclusions and their implications need to be considered in context. Although the scoring process just described might lead to the conclusion that "independent testers are warranted," testers could be introduced in a variety of ways. (1) All quality testing beyond unit testing could be outsourced to an a testing-services vendor. (2) Independent testers could be engaged to oversee the planning and execution of the testing, while internal resources perform all the detailed work. (3) Independent testers could be engaged for all formal testing of units and integration for the critical subsystems, and internal resources could be deployed for noncritical subsystems. The "bottom line" is that each organization needs to develop its own solutions for the inevitable tradeoffs of cost, time, and quality. Hence, the management of quality should be viewed as one more (although a very important) aspect of the overall IT governance process.

---

[4]James A. Whittaker, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software* (January–February 2000), 74-78.

Table 5-1

Choosing In-House or Outsourced Testing

Liability measure	Independent testing is desirable where failure presents a high degree of threat to life or of monetary loss
System complexity	Multiple interoperability (as compared with stand-alone operation) increases risks that can be mitigated by the use of independent testing
Regulatory requirements	Oversight by agencies (FDA, FAA, and NRC) and by regulations (HIPAA) increases the negative consequences of faulty software
Security exposure	Sensitive information that has financial consequences, such as identity theft, increases the risk of financial loss
Nature of users	Novice users (as compared with expert users) need robust systems
Nature of software	Multiple releases (as compared with a one-time release) require robust systems
Nature of use	When external use (as compared with internal use) is intended, the product needs to be robust
Proprietary knowledge	The more information the testers will need about internal users, processes, and systems, the less attractive outsourcing becomes (but the precision needed to communicate with independent testers may offset this factor to some extent)
Reliance on test technology	Complex systems that are intended to have frequent releases require testing with sophisticated tools and methods (e.g., regression testing)
Project timing	Aggressive schedules may require round-the-clock (24-7) testing, which off-shore outsourcing can facilitate

FDA = Food and Drug Administration FAA = Federal Aviation Agency HIPAA = Health Insurance Portability and Accountability Act of 1996 NRC = National Research Council

Other economic factors are to be considered as well. For organizations engaging in aggressive outsourcing of various IT functions ("selective outsourcing"), the outsourcing of the quality and testing group is an interesting opportunity. If one were to prioritize the various IT groups in terms of the need to maintain internal expertise, business analysis and architecture would rank highest; construction, support, and quality are functions that can be easily and cost-effectively outsourced.

5.6 The Auditing Analogy

Within any given organization, auditing functions take place at many levels and are carried out by both internal and external auditors. Beyond the responsibilities of line managers, who ensure that their subordinates are doing their jobs appropriately, a separate organization of internal auditors is established that reports independently to executive management and the board of directors. In order to ensure true quality and accuracy of process, external auditors are engaged and given the key responsibility of verifying the business processes and reported results.

Table 5-2

Factors to Use in Evaluating Outsourcing Testing

Feature	Value	x Weight	Weighted Value
Liability measure			
System complexity			
Regulatory requirements			
Security exposure			
Nature of users			
Nature of software			
Nature of use			
Proprietary knowledge			
Reliance on test technology			
Project timing			
Total			

In the traditional accounting audit, a third-party reviews accounting documentation and declarations in order to ensure full compliance and truthfulness in the financial statements of public companies. This is because the Security and Exchange Commission, which serves as the protector of shareholder rights, relies on the independent auditor to be its unofficial proxy. The challenge of financial oversight over thousands of organizations is mitigated by the release of financial statements that are audited and certified by an independent body that reports not to the organization itself but to an autonomous board of directors.

The same premise of "separation of duties" can be applied to software-development organizations in regulated and unregulated industries alike. In industries such as the health sciences and transportation, software-controlled devices and systems affect life and death, and the consequences of poor quality are extremely high. The use of third-party review increases the probability of regulatory compliance while also sending a message the regulatory bodies and stakeholders of a true commitment to quality.

Just as internal auditors have a direct reporting mechanism to the board of directors, so, too, should internal software testers have a reporting relationship independent of the development organization. The developer is focused on economic and timely software releases, and the tester is focused on software quality—two jobs may potentially be at odds. Furthermore, those persons responsible for producing a product, service, or report should not also be employed to review and declare their accuracy. Similar to the board of directors that reviews financial statements prepared

by outside auditors to insure accuracy, third party testers should have a reporting relationship outside the development organization. This is required so that the development organization can gain the full benefit of what an independent testing organization can deliver in regard to attaining quality and organizationwide process improvements.

This is a key factor in bringing about a quality development organization as well as in ensuring that software systems meet regulatory requirements. An external, independent viewpoint makes it possible for the development organization to make correct decisions about the cost, time, and quality tradeoffs inherent in testing and in decisions about software releases.

Just as the independence requirements of external auditors have been strengthened recently by new government legislation, so, too, the evaluation and compensation of the external testers need to meet appropriate independence criteria. The Sarbanes–Oxley Act of 2002 was initiated in the aftermath of the discovery of accounting irregularities in several large, publicly traded companies. Specifically, it resulted from the discovery that board members and outside auditors had not provided the required financial and operational oversight, at least in part because conflicts of interest had weakened their independence. For example, in many situations the outside audit firm was also providing consulting services, which often generated higher fees and margins than the audit. The act therefore strengthened the definition of independence in the auditing process, so that auditing decisions would not, both in fact and in perception, be influenced by the risk of losing consulting business.

As the need for independent auditing in software development and validation becomes increasingly evident , the Sarbanes–Oxley Act can serve as an ethical and practical compass also for the software industry. The techniques and protective measures used in the financial realm of organizations to ensure effective review surely can be applied to the domain of products and services that affect lives.

5.7 Case Study: The British Post Office's Consignia "Your Guide" Kiosk Project

A case study can illustrate many of the ideas presented about to the value of a comprehensive quality-management program that is based on independent testing: its efficiency, its ability to uncover all types of defects—functional, performance, security, and usability defects—and its value in transforming and improving many preexisting development practices.

In Great Britain, the postal services have been deregulated, with the result that the private sector may participate and therefore compete with the Post Office. This opportunity prompted Consignia (the new owners of the business formerly known as the Post Office) to offer additional services. In a government-sponsored project known as "Your Guide," many stakeholders were involved. The project was piloted between June 16, 2001, and March 1, 2002, in about 270 post offices in and around Leicester, England. "Your Guide" was group of services that were centered on a display in the post offices of a signpost, a free hot-line telephone, a rack of leaflets and, in

most instances, a touch-screen "kiosk." Expert advice was offered by appointment or at drop-in sessions at the post offices as well as over the counter there. The touch screen was the user interface of a computer system custom developed by the Post Office and offered such services as job searching, information about benefits and retirement, free advertising for and local information about businesses and services, among others features. Consignia required a testing solution that would incorporate all of the available features (integration), but that would be focused primarily on the touch-screen system.

Consignia decided to hire an outside group that specialized in testing software systems, and it selected Tescom Software Systems Testing Ltd.[5] to take overall responsibility for testing management, that is, management of a team to test the program. Tescom provided a team to plan, design, and execute tests, from interface testing of the touch screen through to testing of the integration and user acceptance all of the features (channels)—touch screen, leaflets, hot-line and support services, advertisements, and expert advisory sessions. Testing services were provided also to coordinate testing for the management information services (MIS) performance project, which was part of the pilot evaluation. Tescom provided a full testing service that included interface, integration, and acceptance testing, along with load/performance, security, and usability testing.

An important issue Tescom faced is that it was brought into the "Your Guide" project at a late stage, after work had started on physically putting the system together. Hence, the requirements and design phases of the software development life cycle had already been completed, but without the rigorous testing that might have revealed weaknesses, omissions, and flaws. Tescom was able to bring to bear its considerable expertise in developing and executing an efficient and effective test program that brought the project to successful implementation, with the use of four principles.

1. **Identify defects early.** When incidents, issues, or omissions occur, find them as soon as possible to allow the maximum amount of time to fix or resolve them. When failures happen, there need be neither embarrassment nor recrimination; making mistakes will be a characteristic of getting on with the pilot.

2. **Reduce duplication of testing activities.** Full transparency is needed concerning who is testing what; testing roles and responsibilities need to be made clear.

3. **Wherever cost effective and possible, automate testing tasks that will need to be repeated late in development and roll-out,** which will reduce the amount of time spent in testing at the end of the project.

4. **Use testing specialists.** There is no time to learn testing skills during the project.

At the point where Tescom entered the "Your Guide" project, code was being written for the touch-screen system, leaflets were ready to be printed, and the infrastructure was being put in

[5]URL: www.tescom-usa.com

place. Design documentation was scarce and of varying quality. Therefore, while continuously working to improve the quality of the documentation, Tescom decided that use-cases should be the focus for designing the tests. Another consulting firm managed certain test-related processes, such as incident management and system testing. Tescom operated within this framework to assist in making these processes more efficient and robust, setting up the test-director software for tracking defects.

Tescom conducted usability testing in a usability lab in London. Persons typical of the user community—job seekers, working mothers, retired pensioners—were recruited to complete the testing. Program personnel from management, developers, and marketing observed the testing and agreed on the recommendations and subsequent actions.

Tescom completed security testing by dividing the task into three areas: physical security, remote penetration (remote attacks and war dialing), and internal security (vendor-hosted infrastructure security and kiosk security). Regression tests, confidence tests, and some functional tests were automated with the use of Mercury WinRunner. The touch-screen site guide feature, which is a spreadsheet program that shows the static content and navigational pages, was used as the source for a data-driven WinRunner script-generator to automate content testing. The MIS performance project used a rapid application development approach that requires the tester to work closely with the developers. Tescom provided a resource with the technical skills necessary to assure and control the quality of software and to assist in the production of design documentation. A new test-director instance was set up for MIS to use to record and track defects. Thus, Tescom provided the following benefits to Consignia:

- As an independent company, Tescom provided an impartial quality assurance role. The test manager was part of the program's steering group (STAR Chamber). Total control over the testing permitted Tescom to define the testing strategy and control the acceptance criteria for each stage of testing, whether the stages was solely under Tescom's control or was also under the control of a third party.
- Tescom's breadth of service and experience allowed it to highlight issues and in that way assisted in bringing the program to deliver a quality service on time.
- The team that tested the services was reshaped and made more efficient by being focused on delivery; each member had a clearly defined role; and milestones were consistently achieved.
- Tescom implemented a risk-based testing strategy—the risk-and-issues methodology with an agreed-on escalation procedure—to ensure effective use of the budget for testing.

Ultimately, in the view of the project manager of Consignia, without the use of independent testing, the "Your Guide" project could not have been brought to a successful and timely conclusion.

Chapter Six

Benefits, Summary, and Conclusions

Enterprises that invest in proper quality-management programs can expect a variety of benefits that will rapidly repay the costs. First and foremost, new systems that are produced according to a good testing program are far more likely to meet the quantitative and qualitative requirements of the particular enterprise, including functionality, reliability, usability, efficiency, maintainability, and portability. The confidence of the enterprise in the quality of the new systems, as they emerge either from initial development or from major upgrade cycles, will be greater when the developer has used a proper quality-management program.

For the development company, the advantage is that it can rely on the new systems to function as it required, so that the business will avoid such failures as an inability to provide its product to the market and, even more dramatic, the costs of product failure that result in death or serious injury.

A quality-management program enables the developer of the software system to formulate and then achieve specific quality goals, such as progress to higher levels of capability maturity model levels and of the six sigma vision of quality in which quality is equated with the achievement of a rate of no more than 3.4 defects per million opportunities for a product or service transaction. Without having a comprehensive quality program in place that can generate meaningful statistics about the quality of the system being developed, the developer of the system cannot ascertain the current level of the quality of the product or whether that quality is improving.

In a regulated environment, such as the life sciences, a proper testing program can help to ensure that a system will comply with FDA regulations for validation and for 21CFR Part 11 requirements. The incremental cost of validating such systems can be minimized, rather than doubled, as generally assumed by many pharmaceutical companies. In a pragmatic sense, the cost and duration of development will itself be lower, because identifying errors at early phases in the software development life cycle will lower the cost of correcting errors at a later point in development. The development will be measurably less expensive and faster. Robert Lewis[1] calculated that the detection of the most common types of errors during the requirements and design phases, rather than in later phases, could save 9 to 11 percent of the total cost of the system. In the military case Lewis studied, the savings might have exceeded the full cost of an independent testing program.

---

[1]Robert. O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software* (New York: John Wiley, 1992), 278-281.

The challenge, then, is to design and implement a program for quality management. Such a program would be a key benefit of an independent testing organization. The organization could provide expertise in the mechanics of testing as well as objectivity in executing the tests, and it could deliver guidance in establishing the methods, practices, roles, and responsibilities that are required to embed quality management in the development company.

Producing software of high quality requires a quality-management process that needs to be aligned with and integrated into the software development life cycle. The quality of the final product depends on the quality of the developer's practices and deliverables from the start, the specification of requirements phase. First, then, of the best practices is at the earliest point to define the specific, relevant quality metrics and processes to be used in the software development life cycle.

Second, the quality-management program needs to go beyond ensuring that the specific system under development will achieve its quality targets. It needs also to include a rigorous method and process for the continuous improvement of quality. When the product does not achieve the highest quality scores, a root-cause analysis needs to be conducted to determine why a shortfall exists. Was the issue related to training? Did a breakdown in project communications occur? Were the right people for the project not assigned to it? Was the choice of technology inappropriate? There may be hundreds of reasons and factors involved in measuring quality. Identifying those that are most important will allow the developer of the system to modify its processes in order to prevent these reasons and factors from recurring. Defining the metrics used for new or revised processes will make it possible to institutionalize them and make them visible to management, so that persons and projects teams will be accountable for adhering to them.

Third, quality management is a substantial discipline within the broad area of software development. Creating a quality management program that can achieve the objectives of the business enterprise requires professional expertise. Designing a software development life cycle that can balance development resources and quality testing resources, that is efficient and productive, and that uses test automation appropriately is not a simple endeavor. It is also not a trivial job to manage such a program, keep it on track, and ensure that the resources devoted to quality testing are being productively spent. Thus, quality needs to be managed by expert professionals.

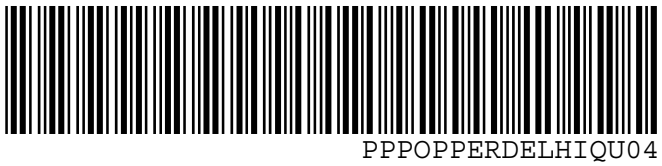
Fourth and last, another best practice is related to the need to have software testers that are independent of the development company. This is necessary for both test definition and for test execution. Defining the appropriate set of tests is best done externally, independently of either intended users or the developers of the system. Independent quality testing, and the perspective it provides, is most useful when applied to the entire software development life cycle, from requirements though integration. It is invaluable during the test-execution phase, when results can be evaluated with rigor and objectivity, and to achieving specific conclusions and recommend-

ations about the need for fundamental rework or redesign, or both, as compared with merely patching defects.

Few companies have a cadre of employees who are dedicated testers, who view themselves as test professionals, keep themselves up to date on the latest techniques and tools to use to automate the testing process, and report directly to senior IT management outside the development company. If a company has such a cadre, it may need third-party testers only to a limited extent. But few organizations devote their resources to testing in this way, and therefore most would benefit substantially from using external testers. When enterprises seek to outsource functions that are not its core competences and which may be less expensive to outsource than to create or purchase internally, then outsourcing testing can be beneficial. As business enterprises focus, with some urgency, on issues of accountability, reliability, integrity, and security, the use of independent testing can enhance the software systems whose successful operation is critical to their financial, operational, and strategic success.

Acronyms

CMM	Capability Maturity Model
DBMS	Data Base Management System
DFSS	Design for Six Sigma
FAA	Federal Aviation Administration
FDA	Food and Drug Administration
GCP	Good Clinical Practices
GLP	Good Laboratory Practices
GMP	Good Manufacturing Practices
HIPAA	Health Insurance Portability and Accountability Act of 1996
ISO	International Standards Organization
IT	Information Technology
IV&V	Independent Validation and Verification
MIS	Management Information Services
NRC	Nuclear Regulatory Commission
QA	Quality Assurance
SDLC	Software Development Life Cycle
SQA	Software Quality Assurance
TQM	Total Quality Management



PPPOPPERDELHIQU04

ISBN-1-879716-88-7