Chapter 1. What Is Software Quality?

Quality must be defined and measured if improvement is to be achieved. Yet, a major problem in quality engineering and management is that the term quality is ambiguous, such that it is commonly misunderstood. The confusion may be attributed to several reasons. First, quality is not a single idea, but rather a multidimensional concept. The dimensions of quality include the entity of interest, the viewpoint on that entity, and the quality attributes of that entity. Second, for any concept there are levels of abstraction; when people talk about quality, one party could be referring to it in its broadest sense, whereas another might be referring to its specific meaning. Third, the term quality is a part of our daily language and the popular and professional uses of it may be very different.

In this chapter we discuss the popular views of quality, its formal definitions by quality experts and their implications, the meaning and specific uses of quality in software, and the approach and key elements of total quality management.

1.1 Quality: Popular Views

A popular view of quality is that it is an intangible trait—it can be discussed, felt, and judged, but cannot be weighed or measured. To many people, quality is similar to what a federal judge once commented about obscenity: "I know it when I see it." Terms such as good quality, bad quality, and quality of life exemplify how people talk about something vague, which they don't intend to define. This view reflects the fact that people perceive and interpret quality in different ways. The implication is that quality cannot be controlled and managed, nor can it be quantified. This view is in vivid contrast to the professional view held in the discipline of quality engineering that quality can, and should, be operationally defined, measured, monitored, managed, and improved.

Another popular view is that quality connotes luxury, class, and taste. Expensive, elaborate, and more complex products are regarded as offering a higher level of quality than their humbler counterparts. Therefore, a Cadillac is a quality car, but a Chevrolet is not, regardless of reliability and repair records; or, a surround-sound hi-fi system is a quality system, but a single-speaker radio is not. According to this view, quality is restricted to a limited class of expensive products with sophisticated functionality and items that have a touch of class. Simple, inexpensive products can hardly be classified as quality products.

1.2 Quality: Professional Views

The misconceptions and vagueness of the popular views do not help the quality improvement effort in the industries. To that end, quality must be described in a workable definition. Crosby (1979) defines quality as "conformance to requirements" and Juran and Gryna (1970) define it as "fitness for use." These two definitions are related and consistent, as we will see later. These definitions of quality have been adopted and used by many quality professionals.

"Conformance to requirements" implies that requirements must be clearly stated such that they cannot be misunderstood. Then, in the development and production process, measurements are taken regularly to determine conformance to those requirements. The nonconformances are regarded as defects—the absence of quality. For example, one requirement (specification) for a certain radio may be that it must be able to receive certain frequencies more than 30 miles away from the source of broadcast. If the radio fails to do so, then it does not meet the quality requirements and should be rejected. By the same token, if a Cadillac conforms to all the requirements of a Cadillac, then it is a quality car. If a Chevrolet conforms to all the requirements of a Chevrolet, then it is also a quality car. The two cars may be very different in style, performance, and economy. But if both measure up to the standards set for them, then both are quality cars.

The "fitness for use" definition takes customers' requirements and expectations into account, which involve whether the products or services fit their uses. Since different customers may use the products in different ways, it means that products must possess multiple elements of fitness for use. According to Juran, each of these elements is a quality characteristic and all of them can be classified into categories known as parameters for fitness for use. The two most important parameters are quality of design and quality of conformance.

Quality of design in popular terminology is known as grades or models, which are related to the spectrum of purchasing power. The differences between grades are the result of intended or designed differences. Using the example of cars again, all automobiles provide to the user the service of transportation. However, models differ in size, comfort, performance, style, economy, and status. In contrast, quality of conformance is the extent to which the product conforms to the intent of the design. In other words, quality of design can be regarded as the determination of requirements and specifications and quality of conformance is conformance to requirements.

The two definitions of quality (conformance to requirements and fitness for use), therefore, are essentially similar. The difference is that the fitness for use concept implies a more significant role for customers' requirements and expectations.

1.2.1 The Role of the Customer

The role of the customer, as it relates to quality, can never be overstated. From a customer's standpoint, quality is the customer's perceived value of the product he or she purchased, based on a variety of variables such as price, performance, reliability, and satisfaction. In Guaspari's book I Know It When I See It (1985 p.77), he discusses quality in the customers' context as follows:

Your customers are in a perfect position to tell you about quality, because that's all they're really buying. They're not buying a product. They're buying your assurances that their expectations for that product will be met.

And you haven't really got anything else to sell them but those assurances. You haven't really got anything else to sell but quality.

From a concept's high-level definition to a product's operational definition, many steps are involved, each of which may be vulnerable to shortcomings. For example, to achieve the state of conformance to requirements, the customers' requirements must be first gathered and analyzed, specifications to meet those requirements must be produced, and the product must be developed and manufactured accordingly. In each phase of the process, errors can occur that will affect the quality of the finished product. The requirements may be erroneous (this is especially the case for software development), the development and manufacturing process may be subject to variables that induce defects, and so forth. From the customer's perspective, satisfaction after the purchase of the product is the ultimate validation that the product conforms to requirements and is fit to use. From the producer's perspective, once requirements are specified, developing and producing the product in accordance with the specifications is the path to achieving quality. Usually, for product quality, the lack of defects and good reliability are the most basic measures.

Because of the two perspectives on quality (customer satisfaction as the ultimate validation of quality and producer's adherence to requirements to achieve quality), the de facto definition of quality consists of two levels. The first is the intrinsic product quality, often operationally limited to the product's defect rate and reliability. This narrow definition is referred to as the "small q" (q for quality). The broader definition of quality includes product quality, process quality, and customer satisfaction, and it is referred to as the "big Q." This two-level approach to the definition of quality is being used in many industries, including the automobile industry, the computer industry (both software and hardware), and the consumer electronics industry.

The two-level concept of quality is supposed to form a closed-loop cycle: customer's wants and needs requirements and specifications products designed, developed, and manufactured in accordance with the requirements, and with continuous focus on process improvement excellent product quality, plus good distribution and service processes total customer satisfaction. However, this was not always the case in many industries, especially before the late 1980s when the modern quality era began. Product requirements were often generated without customer input, and customer satisfaction was not always a factor in business decision making. Although the final products conformed to requirements, they may not have been what the customers wanted. Therefore, the customers' role should be explicitly stated in the definition of quality: conformance to customers' requirements.

1.3 Software Quality

In software, the narrowest sense of product quality is commonly recognized as lack of "bugs" in the product. It is also the most basic meaning of conformance to requirements, because if the software contains too many functional defects, the basic requirement of providing the desired function is not met. This definition is usually expressed in two ways: defect rate (e.g., number of defects per million lines of source code, per function point, or other unit) and reliability (e.g., number of failures per n hours of operation, mean time to failure, or the probability of failure-free operation in a specified time). Customer satisfaction is usually measured by percent satisfied or nonsatisfied (neutral and dissatisfied) from customer satisfaction surveys. To reduce bias, techniques such as blind surveys (the interviewer does not know the customer and the customer does not know the company the interviewer represents) are usually used. In addition to overall customer satisfaction with the software product, satisfaction toward specific attributes is also gauged. For instance, IBM monitors satisfaction with its software products in levels of CUPRIMDSO (capability [functionality], usability, performance, reliability, installability, maintainability, documentation/information, service, and overall). Hewlett-Packard focuses on FURPS (functionality, usability, reliability, performance, and serviceability). Other compa-nies use similar dimensions of software customer satisfaction. Juran calls such attributes quality parameters, or parameters for fitness for use.

To increase overall customer satisfaction as well as satisfaction with various quality attributes, the quality attributes must be taken into account in the planning and design of the software. However, these quality attributes are not always congruous with each other. For example, the higher the functional complexity of the software, the harder it becomes to achieve maintainability. Depending on the type of software and customers, different weighting factors are needed for different quality attributes. For large customers with sophisticated networks and real-time processing, performance and reliability may be the most important attributes. For customers with standalone systems and simple operations, on the other hand, ease of use, installability, and documentation may be more important. Figure 1.1 shows the possible relationships of some quality attributes. Some relationships are mutually supportive, some are negative, and yet others are not clear, depending on the types of customers and applications. For software with a diverse customer set, therefore, setting goals for various quality attributes and to meet customers' requirements is not easy.

Figure 1.1. Interrelationships of Software Attributes—A CUPRIMDA Example

In view of these discussions, the updated definition of quality (i.e., conformance to customers' requirements) is especially relevant to the software industry. It is not surprising that requirements errors constitute one of the major problem categories in software development. According to Jones (1992), 15% or more of all software defects are requirements errors. A development process that does not address requirements quality is bound to produce poor-quality software.

Yet another view of software quality is that of process quality versus end-product quality. From customer requirements to the delivery of software products, the development process is complex and often involves a series of stages, each with feedback paths. In each stage, an intermediate deliverable is produced for an intermediate user—the next stage. Each stage also receives an intermediate deliverable from the preceding stage. Each intermediate deliverable has certain quality attributes that affect the quality of the end product. For instance, Figure 1.2 shows a simplified representation of the most common software development process, the waterfall process.

Figure 1.2. Simplified Representation of the Waterfall Development Process

Intriguingly, if we extend the concept of customer in the definition of quality to include both external and internal customers, the definition also applies to process quality. If each stage of the development process meets the requirements of its intermediate user (the next stage), the end product thus developed and produced will meet the specified requirements. This statement, of course, is an oversimplification of reality, because in each stage numerous factors exist that will affect that stage's ability to fulfill its requirements. To improve quality during development, we need models of the development process, and within the process we need to select and deploy specific methods and approaches, and employ proper tools and technologies. We need measures of the characteristics and quality parameters of the development process and its stages, as well as metrics and models to ensure that the development process is under control and moving toward the product's quality objectives. Quality metrics and models are the focus of this book.

1.4 Total Quality Management

The term Total quality management (TQM) was originally coined in 1985 by the Naval Air Systems Command to describe its Japanese-style management approach to quality improvement. The term has taken on a number of meanings, depending on who is interpreting it and how they are applying it. In general, however, it represents a style of management aimed at achieving long-term success by linking quality and customer satisfaction. Basic to the approach is the creation of a culture in which all members of the organization participate in the improvement of processes, products, and services. Various specific methods for implementing the TQM philosophy are found in the works of Crosby (1979), Deming (1986), Feigenbaum (1961, 1991), Ishikawa (1985), and Juran and Gryna (1970).

Since the 1980s, many U.S. companies have adopted the TQM approach to quality. The Malcolm Baldrige National Quality Award (MBNQA), established by the U.S. government in 1988, highlights the embracing of such a philosophy and management style. The adoption of ISO 9000 as the quality management standard by the European Community and the acceptance of such standards by the U.S. private sector in recent years further illustrates the importance of the quality philosophy in today's business environments. In the computer and electronic industry, examples of successful TQM implementation include Hewlett-Packard's Total Quality Control (TQC), Motorola's Six Sigma Strategy, and IBM's Market Driven Quality. In fact, Motorola won the first MBNQA award (in 1988) and IBM's AS/400 Division in Rochester, Minnesota, won it in 1990.

Hewlett-Packard's TQC focuses on key areas such as management commitment, leadership, customer focus, total participation, and systematic analysis. Each area has strategies and plans to drive the improvement of quality, efficiency, and responsiveness, with the final objective being to achieve success through customer satisfaction (Shores, 1989). In software development, the Software Quality and Productivity Analysis (SQPA) program (Zimmer, 1989) is one of the approaches to improve quality.

Motorola's Six Sigma strategy focuses on achieving stringent quality levels in order to obtain total customer satisfaction. Cycle time reduction and participative management are among the key initiatives of the strategy (Smith, 1989). Six Sigma is not just a measure of the quality level; inherent in the concept are product design improvements and reductions in process variations (Harry and Lawson, 1992). Six Sigma is applied to product quality as well as everything that can be supported by data and measurement.

"Customer is the final arbiter" is the key theme of IBM's Market Driven Quality strategy. The strategy comprises four initiatives: defect elimination, cycle time reduction, customer and business partner satisfaction, and adherence to the Baldrige assessment discipline.

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

Customer focus: The objective is to achieve total customer satisfaction. Customer focus includes studying customers' wants and needs, gathering customers' requirements, and measuring and managing customers' satisfaction.

Process: The objective is to reduce process variations and to achieve continuous process improvement. This element includes both the business process and the product development process. Through process improvement, product quality will be enhanced.

Human side of quality: The objective is to create a companywide quality culture. Focus areas include leadership, management commitment, total participation, employee empowerment, and other social, psychological, and human factors.

Measurement and analysis: The objective is to drive continuous improvement in all quality parameters by the goal-oriented measurement system.

Furthermore, an organization that practices TQM must have executive leadership, must focus on infrastructure, training, and education, and must do strategic quality planning.

Figure 1.3 is a schematic representation of the key elements of TQM. Clearly, measurement and analysis are the fundamental elements for gauging continuous improvement.

Figure 1.3. Key Elements of Total Quality Management

Various organizational frameworks have been proposed to improve quality that can be used to substantiate the TQM philosophy. Specific examples include Plan-Do-Check-Act (Deming, 1986; Shewhart, 1931), Quality Improvement Paradigm/ Experience Factory Organization (Basili, 1985, 1989; Basili and Rombach, 1987, 1988; Basili et al., 1992), Software Engineering Institute (SEI) Capability Maturity Model (Humphrey, 1989; Radice et al., 1985), and Lean Enterprise Management (Womack et al., 1990).

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

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

The SEI Capability Maturity Model is a staged process improvement, based on assessment of key process areas, until you reach level 5, which represents a continuous process improvement. The approach is based on organizational and quality management maturity models developed by Likert (1967) and Crosby (1979), respectively. The goal of the approach is to achieve continuous process improvement via defect prevention, technology innovation, and process change management.

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

Lean Enterprise Management is based on the principle of concentration of production on "value-added" activities and the elimination or reduction of "not-value-added" activities. The approach has been used to improve factory output. The goal is to build software with the minimum necessary set of activities and then to tailor the process to the product's requirements. The approach uses such concepts as technology management, human-centered management, decentralized organization, quality management, supplier and customer integration, and internationalization/ regionalization. Basic to this approach is the assumption that the process can be tailored to classes of problems.

Chapter 2. Software Development Process Models

Software metrics and models cannot be discussed in a vacuum; they must be referenced to the software development process. In this chapter we summarize the major process models being used in the software development community. We start with the waterfall process life-cycle model and then cover the prototyping approach, the spiral model, the iterative development process, and several approaches to the object-oriented development process. Processes pertinent to the improvement of the development process, such as the Cleanroom methodology and the defect prevention process, are also described.

In the last part of the chapter we shift our discussion from specific development processes to the evaluation of development processes and quality management standards. Presented and discussed are the process maturity framework, including the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) and the Software Productivity Research's (SPR) assessment approach, and two bodies of quality standards—the Malcolm Baldrige assessment discipline and ISO 9000—as they relate to software process and quality.

2.1 The Waterfall Development Model

In the 1960s and 1970s software development projects were characterized by massive cost overruns and schedule delays; the focus was on planning and control (Basili and Musa, 1991). The emergence of the waterfall process to help tackle the growing complexity of development projects was a logical event (Boehm, 1976). As Figure 1.2 in Chapter 1 shows, the waterfall process model encourages the development team to specify what the software is supposed to do (gather and define system requirements) before developing the system. It then breaks the complex mission of development into several logical steps (design, code, test, and so forth) with intermediate deliverables that lead to the final product. To ensure proper execution with good-quality deliverables, each step has validation, entry, and exit criteria. This Entry-Task-Validation-Exit (ETVX) paradigm is a key characteristic of the waterfall process and the IBM programming process architecture (Radice et al., 1985).

The divide-and-conquer approach of the waterfall process has several advantages. It enables more accurate tracking of project progress and early identification of possible slippages. It forces the organization that develops the software system to be more structured and manageable. This structural approach is very important for large organizations with large, complex development projects. It demands that the process generate a series of documents that can later be used to test and maintain the system (Davis et al., 1988). The bottom line of this approach is to make large software projects more manageable and delivered on time without cost overrun. Experiences of the past several decades show that the waterfall process is very valuable. Many major developers, especially those who were established early and are involved with systems development, have adopted this process. This group includes commercial corporations, government contractors, and governmental entities. Although a variety of names have been given to each stage in the model, the basic methodologies remain more or less the same. Thus, the system-requirements stages are sometimes called system analysis, customer-requirements gathering and analysis, or user needs analysis; the design stage may be broken down into high-level design and detail-level design; the implementation stage may be called code and debug; and the testing stage may include component-level test, product-level test, and system-level test.

Figure 2.1 shows an implementation of the waterfall process model for a large project. Note that the requirements stage is followed by a stage for architectural design. When the system architecture and design are in place, design and development work for each function begins. This consists of high-level design (HLD), low-level design (LLD), code development, and unit testing (UT). Despite the waterfall concept, parallelism exists because various functions can proceed simultaneously. As shown in the figure, the code development and unit test stages are also implemented iteratively. Since UT is an integral part of the implementation stage, it makes little sense to separate it into another formal stage. Before the completion of the HLD, LLD, and code, formal reviews and inspections occur as part of the validation and exit criteria. These inspections are called I0, I1, and I2 inspections, respectively. When the code is completed and unit tested, the subsequent stages are integration, component test, system test, and early customer programs. The final stage is release of the software system to customers.

Figure 2.1. An Example of the Waterfall Process Model

The following sections describe the objectives of the various stages from highlevel design to early customer programs.

High-Level Design

High-level design is the process of defining the externals and internals from the perspective of a component. Its objectives are as follows:

Develop the external functions and interfaces, including:

external user interfaces

application programming interfaces

system programming interfaces: intercomponent interfaces and data structures.

Design the internal component structure, including intracomponent interfaces and data structures.

Ensure all functional requirements are satisfied.

Ensure the component fits into the system/product structure.

Ensure the component design is complete.

Ensure the external functions can be accomplished—"doability" of

requirements.

Low-Level Design

Low-level design is the process of transforming the HLD into more detailed designs from the perspective of a part (modules, macros, includes, and so forth). Its objectives are as follows:

Finalize the design of components and parts (modules, macros, includes) within a system or product.

Complete the component test plans.

Give feedback about HLD and verify changes in HLD.

Code Stage

The coding portion of the process results in the transformation of a function's LLD to completely coded parts. The objectives of this stage are as follows:

Code parts (modules, macros, includes, messages, etc.).

Code component test cases.

Verify changes in HLD and LLD.

Unit Test

The unit test is the first test of an executable module. Its objectives are as follows:

Verify the code against the component's

high-level design and

low-level design.

Execute all new and changed code to ensure

all branches are executed in all directions,

logic is correct, and

data paths are verified.

Exercise all error messages, return codes, and response options.

Give feedback about code, LLD, and HLD.

The level of unit test is for verification of limits, internal interfaces, and logic and data paths in a module, macro, or executable include. Unit testing is performed on nonintegrated code and may require scaffold code to construct the proper environment.

Component Test

Component tests evaluate the combined software parts that make up a component after they have been integrated into the system library. The objectives of this test are as follows:

Test external user interfaces against the component's design documentation— user requirements.

Test intercomponent interfaces against the component's design documentation.

Test application program interfaces against the component's design documentation.

Test function against the component's design documentation.

Test intracomponent interfaces (module level) against the component's design documentation.

Test error recovery and messages against the component's design documentation.

Verify that component drivers are functionally complete and at the acceptable quality level.

Test the shared paths (multitasking) and shared resources (files, locks, queues, etc.) against the component's design documentation.

Test ported and unchanged functions against the component's design documentation.

System-Level Test

The system-level test phase comprises the following tests:

System test

System regression test

System performance measurement test

Usability tests

The system test follows the component tests and precedes system regression tests. The system performance test usually begins shortly after system testing starts and proceeds throughout the system-level test phase. Usability tests occur throughout the development process (i.e., prototyping during design stages, formal usability testing during system test period).

System test objectives

Ensure software products function correctly when executed concurrently and in stressful system environments.

Verify overall system stability when development activity has been completed for all products.

System regression test objective

Verify that the final programming package is ready to be shipped to external customers.

Make sure original functions work correctly after functions were added to the system.

System performance measurement test objectives

Validate the performance of the system.

Verify performance specifications.

Provide performance information to marketing.

Establish base performance measurements for future releases.

Usability tests objective

Verify that the system contains the usability characteristics required for the intended user tasks and user environment.

Early Customer Programs

The early customer programs (ECP) include testing of the following support structures to verify their readiness:

Service structures

Development fix support

Electronic customer support

Market support

Ordering, manufacturing, and distribution

In addition to these objectives, a side benefit of having production systems installed in a customer's environment for the ECP is the opportunity to gather customers' feedback so developers can evaluate features and improve them for future releases. Collections of such data or user opinion include:

Product feedback: functions offered, ease of use, and quality of online documentation

Installability of hardware and software

Reliability

Performance (measure throughput under the customer's typical load)

System connectivity

Customer acceptance

As the preceding lists illustrate, the waterfall process model is a disciplined approach to software development. It is most appropriate for systems development characterized by a high degree of complexity and interdependency. Although expressed as a cascading waterfall, parallelism and some amount of iteration among process phases often exist in actual implementation. During this process, the focus should be on the intermediate deliverables (e.g., design document, interface rules, test plans, and test cases) rather than on the sequence of activities for each development phase. In other words, it should be entity-based instead of step-by-step based. Otherwise the process could become too rigid to be efficient and effective.

**2.2 The Prototyping Approach**

The first step in the waterfall model is the gathering and analysis of customers' requirements. When the requirements are defined, the design and development work begins. The model assumes that requirements are known, and that once requirements are defined, they will not change or any change will be insignificant. This may well be the case for system development in which the system's purpose and architecture are thoroughly investigated. However, if requirements change significantly between the time the system's specifications are finalized and when the product's development is complete, the waterfall may not be the best model to deal with the resulting problems. Sometimes the requirements are not even known. In the past, various software process models have been proposed to deal with customer feedback on the product to ensure that it satisfied the requirements. Each of these models provides some form of prototyping, of either a part or all of the system. Some of them build prototypes to be thrown away; others evolve the prototype over time, based on customer needs.

A prototype is a partial implementation of the product expressed either logically or physically with all external interfaces presented. The potential customers use the prototype and provide feedback to the development team before full-scale development begins. Seeing is believing, and that is really what prototyping intends to achieve. By using this approach, the customers and the development team can clarify requirements and their interpretation.

As Figure 2.2 shows, the prototyping approach usually involves the following steps:

Gather and analyze requirements.

Do a quick design.

Build a prototype.

Customers evaluate the prototype.

Refine the design and prototype.

If customers are not satisfied with the prototype, loop back to step 5.

If customers are satisfied, begin full-scale product development.

Figure 2.2. The Prototyping Approach

The critical factor for success of the prototyping approach is quick turnaround in designing and building the prototypes. Several technologies can be used to achieve such an objective. Reusable software parts could make the design and implementation of prototypes easier. Formal specification languages could facilitate the generation of executable code (e.g., the Z notation and the Input/Output Requirements Language (IORL) (Smith and Wood, 1989; Wing, 1990)). Fourth-generation lan-guages and technologies could be extremely useful for prototyping in the graphical user interface (GUI) domain. These technologies are still emerging, however, and are used in varying degrees depending on the specific characteristics of the projects.

The prototyping approach is most applicable to small tasks or at the subsystem level. Prototyping a complete system is difficult. Another difficulty with this approach is knowing when to stop iterating. In practice, the method of time boxing is being used. This method involves setting arbitrary time limits (e.g., three weeks) for each activity in the iteration cycle and for the entire iteration and then assessing progress at these checkpoints.

Rapid Throwaway Prototyping

The rapid throwaway prototyping approach of software development, made popular by Gomaa and Scott (1981), is now used widely in the industry, especially in application development. It is usually used with high-risk items or with parts of the system that the development team does not understand thoroughly. In this approach, "quick and dirty" prototypes are built, verified with customers, and thrown away until a satisfactory prototype is reached, at which time full-scale development begins.

Evolutionary Prototyping

In the evolutionary prototyping approach, a prototype is built based on some known requirements and understanding. The prototype is then refined and evolved instead of thrown away. Whereas throwaway prototypes are usually used with the aspects of the system that are poorly understood, evolutionary prototypes are likely to be used with aspects of the system that are well understood and thus build on the development team's strengths. These prototypes are also based on prioritized requirements, sometimes referred to as "chunking" in application development (Hough, 1993). For complex applications, it is not reasonable or economical to expect the prototypes to be developed and thrown away rapidly.

2.3 The Spiral Model

The spiral model of software development and enhancement, developed by Boehm (1988), is based on experience with various refinements of the waterfall model as applied to large government software projects. Relying heavily on prototyping and risk management, it is much more flexible than the waterfall model. The most comprehensive application of the model is the development of the TRW Software Productivity System (TRW-SPS) as described by Boehm. The spiral concept and the risk management focus have gained acceptance in software engineering and project management in recent years.

Figure 2.3 shows Boehm's spiral model. The underlying concept of the model is that each portion of the product and each level of elaboration involves the same sequence of steps (cycle). Starting at the center of the spiral, one can see that each development phase (concept of operation, software requirements, product design, detailed design, and implementation) involves one cycle of the spiral. The radial dimension in Figure 2.3 represents the cumulative cost incurred in accomplishing the steps. The angular dimension represents the progress made in completing each cycle of the spiral. As indicated by the quadrants in the figure, the first step of each cycle of the spiral is to identify the objectives of the portion of the product being elaborated, the alternative means of implementation of this portion of the product, and the constraints imposed on the application of the alternatives. The next step is to evaluate the alternatives relative to the objectives and constraints, to identify the associated risks, and to resolve them. Risk analysis and the risk-driven approach, therefore, are key characteristics of the spiral model, in contrast to the document-driven approach of the waterfall model.

**Figure 2.3. Spiral Model of the Software Process**

From "A Spiral Model of Software Development and Enhancement," by B. W. Boehm. IEEE Computer (May): 61–72. © 1988 IEEE. Reprinted with permission.

In this risk-driven approach, prototyping is an important tool. Usually prototyping is applied to the elements of the system or the alternatives that present the higher risks. Unsatisfactory prototypes can be thrown away; when an operational prototype is in place, implementation can begin. In addition to prototyping, the spiral model uses simulations, models, and benchmarks in order to reach the best alternative. Finally, as indicated in the illustration, an important feature of the spiral model, as with other models, is that each cycle ends with a review involving the key members or organizations concerned with the product.

For software projects with incremental development or with components to be developed by separate organizations or individuals, a series of spiral cycles can be used, one for each increment or component. A third dimension could be added to Figure 2.3 to represent the model better.

Boehm (1988) provides a candid discussion of the advantages and disadvantages of the spiral model. Its advantages are as follows:

Its range of options accommodates the good features of existing software process models, whereas its risk-driven approach avoids many of their difficulties. This is the primary advantage. Boehm also discusses the primary conditions under which this model becomes equivalent to other process models such as the waterfall model and the evolutionary prototype model.

It focuses early attention on options involving the reuse of existing software. These options are encouraged because early identification and evaluation of alternatives is a key step in each spiral cycle. This model accommodates preparation for life-cycle evolution, growth, and changes of the software product.

It provides a mechanism for incorporating software quality objectives into software product development.

It focuses on eliminating errors and unattractive alternatives early.

It does not involve separate approaches for software development and software enhancement.

It provides a viable framework for integrating hardware-software system development. The risk-driven approach can be applied to both hardware and software.

On the other hand, difficulties with the spiral model include the following:

Matching to contract software: Contract software relies heavily on control, checkpoint, and intermediate deliverables for which the waterfall model is good. The spiral model has a great deal of flexibility and freedom and is, therefore, more suitable for internal software development. The challenge is how to achieve the flexibility and freedom prescribed by the spiral model without losing accountability and control for contract software.

Relying on risk management expertise: The risk-driven approach is the back-bone of the model. The risk-driven specification addresses high-risk elements in great detail and leaves low-risk elements to be elaborated in later stages. However, an inexperienced team may also produce a specification just the opposite: a great deal of detail for the well-understood, low-risk elements and little elaboration of the poorly understood, high-risk elements. In such a case, the project may fail and the failure may be discovered only after major resources have been invested. Another concern is that a risk-driven specification is people dependent. In the case where a design produced by an expert is to be implemented by nonexperts, the expert must furnish additional documentation.

Need for further elaboration of spiral steps: The spiral model describes a flexible and dynamic process model that can be used to its fullest advantage by experienced developers. For nonexperts and especially for large-scale projects, however, the steps in the spiral must be elaborated and more specifically defined so that consistency, tracking, and control can be achieved. Such elaboration and control are especially important in the area of risk analysis and risk management.

**Chapter 3. Fundamentals of Measurement Theory**

This chapter discusses the fundamentals of measurement theory. We outline the relationships among theoretical concepts, definitions, and measurement, and describe some basic measures that are used frequently. It is important to distinguish the levels of the conceptualization proces s, from abstract concepts, to definitions that are used operationally, to actual measurements. Depending on the concept and the operational definition derived from it, different levels of measurement may be applied: nominal scale, ordinal scale, interval scale, and ratio scale. It is also beneficial to spell out the explicit differences among some basic measures such as ratio, proportion, percentage, and rate. Significant amounts of wasted effort and resources can be avoided if these fundamental measurements are well understood.

**Chapter 4. Software Quality Metrics Overview**

**S**oftware metrics can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the inprocess quality metrics of a project are both process metrics and project metrics.

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of the product. Software quality metrics can be divided further into end-product quality metrics and in-process quality metrics. The essence of software quality engineering is to investigate the relationships among in-process metrics, project characteristics, and end-product quality, and, based on the findings, to engineer improvements in both process and product quality. Moreover, we should view quality from the entire software life-cycle perspective and, in this regard, we should include metrics that measure the quality level of the maintenance process as another category of software quality metrics. In this chapter we discuss several metrics in each of three groups of software quality metrics: product quality, in-process quality, and maintenance quality. In the last sections we also describe the key metrics used by several major software developers and discuss software metrics data collection.

We then focus on measurement quality. We discuss the most important issues in measurement quality, namely, reliability and validity, and their relationships with measurement errors. We then discuss the role of correlation in observational studies and the criteria for causality.

Chapter 5. Applying the Seven Basic Quality Tools in Software Development

The basic statistical tools for quality control promoted by Ishikawa (1989) are widely used in manufacturing productions. They have indeed become an integral part of the quality control literature, and have been known as Ishikawa's seven basic tools. This chapter describes the application of these tools for process and quality control in software development. There are many ways to analyze software metrics; the applications of Ishikawa's seven tools represent a set of basic operations. Keep in mind that these statistical tools are for process and quality control at the project and organization level and, hence, are useful for project leaders and process experts. In contrast, they do not provide specific information to software developers on how to improve the quality of their designs or implementation. Also, because not all these tools are equally useful for small projects where statistical patterns of parameters of the development process are less obvious, the benefits of statistics may not be realized. The box at the end of the chapter offers specific recommendations for small teams. In addition, although the benefits of these tools have long been proved in manufacturing operations, their use and roles in software development has not been widely recognized. For instance, the use of control charts in manufacturing production can ensure a certain end-product quality once the process is defined and the control limits are set. In software development, however, the process is complex and involves a high degree of creativity and mental activity. It is extremely difficult, if not impossible, to define the process capability of software development in statistical terms. Therefore, achieving statistical process control in software development may mean a lot more than control charting. It may require, for example, new development technology, CASE tools, and the use of defect models and reliability estimating techniques. However, good use of the seven basic tools can lead to positive long-term results for process improvement and quality management in software development.

The following sections begin with a brief description of the tools, followed by a discussion of each tool with examples of its applications. Where appropriate, the influences of these tools on process improvement and on decision making are also described. The examples are either from software engineering literature or from software projects developed at IBM in Rochester, Minnesota. In addition to the seven basic tools, we discuss the relations diagram, which is effective for small team brainstorming and particularly useful in displaying cause-and-effect relationships.

**Chapter 6. Defect Removal Effectiveness**

The concept of defect removal effectiveness and its measurement are central to software development. Defect removal is one of the top expenses in any software project and it greatly affects schedules. Effective defect removal can lead to reductions in the development cycle time and good product quality. For improvements in quality, productivity, and cost, as well as schedule, it is important to use better defect prevention and removal technologies to maximize the effectiveness of the project. It is important for all projects and development organizations to measure the effectiveness of their defect removal processes.

In Chapter 4 we briefly touched on the metrics of defect removal effectiveness and fault containment. In this chapter we elaborate on the concept, its measurements, and its use in the phase-based defect removal model. After a brief literature review, we take a closer look at the defect injection and removal activities of the phases of a typical development process. Using a matrix approach to cross-tabulate defect data in terms of defect origin and phase of defect discovery (where found), we provide a detailed example of calculating the values of the overall defect removal effectiveness, the inspection effectiveness, the test effectiveness as well as the phase-specific defect removal effectiveness. We then establish the formulas of these indexes based on the defect origin/where found matrix. Next we elaborate the role of defect removal effectiveness in quality planning with more examples. We discuss the cost effectiveness of phase defect removal and also the defect removal effectiveness levels in the context of the capability maturity model (CMM) before summarizing the chapter.

Before we begin, a point on terminology is in order. Some writers use the terms defect removal efficiency, error detection efficiency, fault containment, defect removal effectiveness, and the like. In this book we prefer the term effectiveness rather than efficiency. Efficiency implies the element of time, effectiveness is related to the extent of impact and we think the latter is more appropriate. In the following sections we may sometimes use the two terms interchangeably, especially when we refer to the definitions and metrics of other writers.