CHAPTER 1

SOFTWARE QUALITY CONCEPTS

1.1 WHAT IS QUALITY

The American Heritage Dictionary defines quality as “a characteristic or attribute of something.” Quality is defined in the International Organization for Standardization (ISO) publications as the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.

Quality is a more intriguing concept than it seems to be. The meaning of the term “Quality” has evolved over time as many concepts were developed to improve product or service quality, including total quality management (TQM), Malcolm Baldrige National Quality Award, Six Sigma, quality circles, theory of constraints (TOC), Quality Management Systems (ISO 9000 and ISO 13485), axiomatic quality (El-Haik, 2005), and continuous improvement. The following list represents the various interpretations of the meaning of quality:

- “Quality: an inherent or distinguishing characteristic, a degree or grade of excellence” (American Heritage Dictionary, 1996).
- “Conformance to requirements” (Crosby, 1979).
- “Fitness for use” (Juran & Gryna, 1988).
- “Degree to which a set of inherent characteristic fulfills requirements” ISO 9000.
SOFTWARF QUALITY CONCEPTS

- “Value to some person” (Weinberg).
- “The loss a product imposes on society after it is shipped” (Taguchi).
- “The degree to which the design vulnerabilities do not adversely affect product performance” (El-Haik, 2005).

Quality is a characteristic that a product or service must have. It refers to the perception of the degree to which the product or service meets the customer’s expectations. Quality has no specific meaning unless related to a specific function or measurable characteristic. The dimensions of quality refer to the measurable characteristics that the quality achieves. For example, in design and development of a medical device:

- Quality supports safety and performance.
- Safety and performance supports durability.
- Durability supports flexibility.
- Flexibility supports speed.
- Speed supports cost.

You can easily build the interrelationship between quality and all aspects of product characteristics, as these characteristics act as the qualities of the product. However, not all qualities are equal. Some are more important than others. The most important qualities are the ones that customers want most. These are the qualities that products and services must have. So providing quality products and services is all about meeting customer requirements. It is all about meeting the needs and expectations of customers.

When the word “quality” is used, we usually think in terms of an excellent design or service that fulfills or exceeds our expectations. When a product design surpasses our expectations, we consider that its quality is good. Thus, quality is related to perception. Conceptually, quality can be quantified as follows (El-Haik & Roy, 2005):

\[ Q = \frac{\sum P}{\sum E} \]  

where \( Q \) is quality, \( P \) is performance, and \( E \) is an expectation.

In a traditional manufacturing environment, conformance to specification and delivery are the common quality items that are measured and tracked. Often, lots are rejected because they do not have the correct documentation supporting them. Quality in manufacturing then is conforming product, delivered on time, and having all of the supporting documentation. In design, quality is measured as consistent conformance to customer expectations.
In general, quality\(^2\) is a fuzzy linguistic variable because quality can be very subjective. What is of a high quality to someone might not be a high quality to another. It can be defined with respect to attributes such as cost or reliability. It is a degree of membership of an attribute or a characteristic that a product or software can or should have. For example, a product should be reliable, or a product should be both reliable and usable, or a product should be reliable or repairable. Similarly, software should be affordable, efficient, and effective. These are some characteristics that a good quality product or software must have. In brief, quality is a desirable characteristic that is subjective. The desired qualities are the ones that satisfy the functional and nonfunctional requirements of a project. Figure 1.1 shows a possible membership function, \(\mu(X)\), for the affordable software with respect to the cost \((X)\).

When the word “quality” is used in describing a software application or any product, it implies a product or software program that you might have to pay more for or spend more time searching to find.

### 1.2 QUALITY, CUSTOMER NEEDS, AND FUNCTIONS

The quality of a software product for a customer is a product that meets or exceeds requirements or expectations. Quality can be achieved through many levels (Braude,

---

1. where \(K\) is the max cost value of the software after which the software will be not be affordable \(\mu(K) = 0\).
2. J. M. Juran (1988) defined quality as “fitness for use.” However, other definitions are widely discussed. Quality as “conformance to specifications” is a position that people in the manufacturing industry often promote. Others promote wider views that include the expectations that the product or service being delivered 1) meets customer standards, 2) meets and fulfills customer needs, 3) meets customer expectations, and 4) will meet unanticipated future needs and aspirations.
SOFTWARE QUALITY CONCEPTS

One level for attaining quality is through inspection, which can be done through a team-oriented process or applied to all stages of the software process development. A second level for attaining quality is through formal methods, which can be done through mathematical techniques to prove that the software does what it is meant to do or by applying those mathematical techniques selectively. A third level for attaining quality is through testing, which can be done at the component level or at the application level. A fourth level is through project control techniques, which can be done through predicting the cost and schedule of the project or by controlling the artifacts of the project (scope, versions, etc.). Finally, the fifth level we are proposing here is designing for quality at the Six Sigma level, a preventive and proactive methodology, hence, this book.

A quality function should have the following properties (Braude, 2001):

- Satisfies clearly stated functional requirements
- Checks its inputs; reacts in predictable ways to illegal inputs
- Has been inspected exhaustively in several independent ways
- Is thoroughly documented
- Has a confidently known defect rate, if any

The American Society for Quality (ASQ) defines quality as follows: “A subjective term for which each person has his or her own definition.” Several concepts are associated with quality and are defined as follows:

- **Quality Assurance**: Quality assurance (QA) is defined as a set of activities whose purpose is to demonstrate that an entity meets all quality requirements usually after the fact (i.e., mass production). We will use QA in the Verify & Validate phase of the Design For Six Sigma (DFSS) process in the subsequent chapters. QA activities are carried out to inspire the confidence of both customers and managers that all quality requirements are being met.
- **Quality Audits**: Quality audits examine the elements of a quality management system to evaluate how well these elements comply with quality system requirements.
- **Quality Control**: Quality control is defined as a set of activities or techniques whose purpose is to ensure that all quality requirements are being met. To achieve this purpose, processes are monitored and performance problems are solved.
- **Quality Improvement**: Quality improvement refers to anything that enhances an organization’s ability to meet quality requirements.
- **Quality Management**: Quality management includes all the activities that managers carry out in an effort to implement their quality policy. These activities

---

include quality planning, quality control, quality assurance, and quality improvement.

- **Quality Management System (QMS):** A QMS is a web of interconnected processes. Each process uses resources to turn inputs into outputs. And all of these processes are interconnected by means of many input–output relationships. Every process generates at least one output, and this output becomes an input for another process. These input–output relationships glue all of these processes together—that’s what makes it a system. A quality manual documents an organization’s QMS. It can be a paper manual or an electronic manual.

- **Quality Planning:** Quality planning is defined as a set of activities whose purpose is to define quality system policies, objectives, and requirements, and to explain how these policies will be applied, how these objectives will be achieved, and how these requirements will be met. It is always future oriented. A quality plan explains how you intend to apply your quality policies, achieve your quality objectives, and meet your quality system requirements.

- **Quality Policy:** A quality policy statement defines or describes an organization’s commitment to quality.

- **Quality Record:** A quality record contains objective evidence, which shows how well a quality requirement is being met or how well a quality process is performing. It always documents what has happened in the past.

- **Quality Requirement:** A quality requirement is a characteristic that an entity must have. For example, a customer may require that a particular product (entity) achieve a specific dependability score (characteristic).

- **Quality Surveillance:** Quality surveillance is a set of activities whose purpose is to monitor an entity and review its records to prove that quality requirements are being met.

- **Quality System Requirement:** A quality is a characteristic. A system is a set of interrelated processes, and a requirement is an obligation. Therefore, a quality system requirement is a characteristic that a process must have.

### 1.3 QUALITY, TIME TO MARKET, AND PRODUCTIVITY

The time to market of a software product is how fast a software company can introduce a new or improved software products and services to the market. It is very important for a software company to introduce their products in a timely manner without reducing the quality of their products. The software company that can offer their product faster without compromising quality achieve a tremendous competitive edge with respect to their competitors.

There are many techniques to reduce time to market, such as (El-Haik, 2005):

- Use the proper software process control technique(s), which will reduce the complexity of the software product.
6 SOFTWARE QUALITY CONCEPTS

- Concurrency: Encouraging multitasking and parallelism
- Use the Carnegie Mellon Personal Software Process (PSP) and Team Software Process (TSP) with DFSS (El-Haik & Roy, 2005)
- Project management: Tuned for design development and life-cycle management

Using these techniques and methods would increase the quality of the software product and would speed up the production cycle, which intern reduces time to market the product.

1.4 QUALITY STANDARDS

Software system quality standards according to the IEEE Computer Society on Software Engineering Standards Committee can be an object or measure of comparison that defines or represents the magnitude of a unit. It also can be a characterization that establishes allowable tolerances or constraints for categories of items. Also it can be a degree or level of required excellence or attainment.

Software quality standards define a set of development criteria that guide the way software is engineered. If the criteria are not followed, quality can be affected negatively. Standards sometimes can negatively impact quality because it is very difficult to enforce it on actual program behavior. Also standards used to inappropriate software processes may reduce productivity and, ultimately, quality.

Software system standards can improve quality through many development criteria such as preventing idiosyncrasy (e.g., standards for primitives in programming languages) and repeatability (e.g., repeating complex inspection processes). Other ways to improve software quality includes preventive mechanisms such as Design for Six Sigma (design it right the first time), consensus wisdom (e.g., software metrics), cross-specialization (e.g., software safety standards), customer protection (e.g., quality assurance standards), and badging (e.g., capability maturity model [CMM] levels).

There are many standards organizations. Table 1.1 shows some of these standard organizations.

Software engineering process technology (SEPT) has posted the most popular software Quality standards. Table 1.2 shows the most popular software Quality standards.

1.5 SOFTWARE QUALITY ASSURANCE AND STRATEGIES

Professionals in any field must learn and practice the skills of their professions and must demonstrate basic competence before they are permitted to practice their professions. This is not the case with the software engineering profession (Watts,

TABLE 1.1 Shows Some Standard Organizations

<table>
<thead>
<tr>
<th>Organization</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute (does not itself make standards but approves them)</td>
</tr>
<tr>
<td>AIAA</td>
<td>American Institute of Aeronautics and Astronautics</td>
</tr>
<tr>
<td>EIA</td>
<td>Electronic Industries Association</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electro technical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers Computer Society Software Engineering Standards Committee</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
</tbody>
</table>

1997). Most software engineers learn the skills they need on the job, and this is not only expensive and time consuming, but also it is risky and produces low-quality products.

The work of software engineers has not changed a lot during the past 30 years (Watts, 1997) even though the computer field has gone through many technological advances. Software engineers uses the concept of modular design. They spend a large portion of their time trying to get these modules to run some tests. Then they test and integrate them with other modules into a large system. The process of integrating and testing is almost totally devoted to finding and fixing more defects. Once the software product is deployed, then the software engineers spend more time fixing the defects reported by the customers. These practices are time consuming, costly, and retroactive in contrast to DFSS. A principle of DFSS quality is to build the product right the first time.

The most important factor in software quality is the personal commitment of the software engineer to developing a quality product (Watts, 1997). The DFSS process can produce quality software systems through the use of effective quality and design methods such as axiomatic design, design for X, and robust design, to name few.

The quality of a software system is governed by the quality of its components. Continuing with our fuzzy formulation (Figure 1.1), the overall quality of a software system ($\mu_{\text{Quality}}$) can be defined as

$$\mu_{\text{Quality}} = \min(\mu_{Q1}, \mu_{Q2}, \mu_{Q3}, \ldots, \mu_{Qn})$$

where $\mu_{Q1}, \mu_{Q2}, \mu_{Q3}, \ldots, \mu_{Qn}$ are the quality of the $n$ parts (modules) that makes up the software system, which can be assured by the QA function.

QA includes the reviewing, auditing, and reporting processes of the software product. The goal of quality assurance is to provide management (Pressman, 1997) with the data needed to inform them about the product quality so that the management can control and monitor a product’s quality. Quality assurance does apply throughout a software design process. For example, if the waterfall software design process is followed, then QA would be included in all the design phases (requirements and analysis, design, implementation, testing, and documentation). QA will be included in the requirement and analysis phase through reviewing the functional and
### TABLE 1.2 Shows the Most Popular Software Quality Standards

<table>
<thead>
<tr>
<th>Quality Standard</th>
<th>Name and Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIAA R-013</td>
<td>Recommended Practice for Software Reliability</td>
</tr>
<tr>
<td>ASME NQA-1</td>
<td>Quality Assurance Requirements for Nuclear Facility Applications</td>
</tr>
<tr>
<td>EIA/IS 632</td>
<td>Systems Engineering</td>
</tr>
<tr>
<td>IEC 60880</td>
<td>Software for Computers in the Safety Systems of Nuclear Power Stations</td>
</tr>
<tr>
<td>IEC 61508</td>
<td>Functional Safety Systems</td>
</tr>
<tr>
<td>IEC 62304</td>
<td>Medical Device Software—Software Life Cycle Processes</td>
</tr>
<tr>
<td>IEEE 1058.1–1987</td>
<td>Software Project Management Plans</td>
</tr>
<tr>
<td>IEEE Std 730</td>
<td>Software Quality Assurance Plans</td>
</tr>
<tr>
<td>IEEE Std 730.1</td>
<td>Guide for Software Assurance Planning</td>
</tr>
<tr>
<td>IEEE Std 982.1</td>
<td>Standard Dictionary of Measures to Produce Reliable Software</td>
</tr>
<tr>
<td>IEEE Std 1059–1993</td>
<td>Software Verification and Validation Plans</td>
</tr>
<tr>
<td>IEEE Std 1061</td>
<td>Standard for a Software Quality Metrics Methodology</td>
</tr>
<tr>
<td>IEEE Std 16085</td>
<td>Software Life Cycle Processes—Risk Management</td>
</tr>
<tr>
<td>ISO/IEC 2382-7:1989</td>
<td>Vocabulary—part 7: Computer Programming</td>
</tr>
<tr>
<td>ISO/IEC 8631:1989</td>
<td>Program Constructs and Conventions for their Representation</td>
</tr>
<tr>
<td>ISO/IEC 12119</td>
<td>Information Technology—Software Packages—Quality Requirements and Testing</td>
</tr>
<tr>
<td>ISO/IEC 14102</td>
<td>Guideline For the Evaluation and Selection of CASE Tools</td>
</tr>
<tr>
<td>ISO/IEC 14598-1</td>
<td>Information Technology—Evaluation of Software Products—General Guide</td>
</tr>
<tr>
<td>ISO/IEC WD 15288</td>
<td>System Life Cycle Processes</td>
</tr>
<tr>
<td>ISO/IEC 25030</td>
<td>Software Engineering—Software Product Quality Requirements and Evaluation (SQuaRE)—Quality Requirements</td>
</tr>
</tbody>
</table>
nonfunctional requirements, reviewing for conformance to organizational policy, re-
views for configuration management plans, standards, and so on. QA in the design
phase may include reviews, inspections, and tests. QA would be able to answer ques-
tions like, “Does the software design adequately meet the quality required by the
management?” QA in the implementation phase may include a review provision for
QA activities, inspections, and testing. QA would be able to answer questions like,
“Have technical disciplines properly performed their roles as part of the QA activ-
ity?” QA in the testing phase would include reviews, and several testing activities.
QA in the maintenance phase could include reviews, inspections, and tests as well.
The QA engineer serves as the customer’s in-house representative (Pressman, 1997).
The QA engineer usually is involved with the inspection process. Ideally, QA should
(Braude, 2001) be performed by a separate organization (independent) or engineers
can perform QA functions on each other’s work.

The ANSI/IEEE Std 730-1984 and 983-1986 software quality assurance plans provide a road map for instituting software quality assurance. Table 1.3 shows the
ANSI/IEEE Std 730-1984 and 983-1986 software quality assurance plans. The plans
serve as a template for the QA activates that are instituted for each software project.

The QA activities performed by software engineering team and the QA group are
controlled by the plans. The plans identify the following (Pressman, 1997):

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by the QA group
- Amount of feedback provided to the software project team

To be more precise in measuring the quality of a software product, statistical quality
assurance methods have been used. The statistical quality assurance for software
products implies the following steps (Pressman, 1997):

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its cause.
3. Using the Pareto principle, the 20% of the vital causes of errors that produce
   80% of the defects should be isolated.
4. Once the vital causes have been identified, the problems that have caused the
defects should be corrected.

1.6 SOFTWARE QUALITY COST

Quality is always deemed to have a direct relationship to cost—the higher the quality
standards, the higher the cost. Or so it seems. Quality may in fact have an inverse

TABLE 1.3 ANSI/IEEE Std 730-1984 and 983-1986 Software Quality Assurance Plans

<table>
<thead>
<tr>
<th>Section</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Purpose of the plan</td>
<td></td>
</tr>
<tr>
<td>II. References</td>
<td></td>
</tr>
<tr>
<td>III. Management</td>
<td></td>
</tr>
<tr>
<td>a. Organization</td>
<td></td>
</tr>
<tr>
<td>b. Tasks</td>
<td></td>
</tr>
<tr>
<td>c. Responsibilities</td>
<td></td>
</tr>
<tr>
<td>IV. Documentation</td>
<td></td>
</tr>
<tr>
<td>a. Purpose</td>
<td></td>
</tr>
<tr>
<td>b. Required software engineering documents</td>
<td></td>
</tr>
<tr>
<td>c. Other documents</td>
<td></td>
</tr>
<tr>
<td>V. Standards, practices, and conventions</td>
<td></td>
</tr>
<tr>
<td>a. Purpose</td>
<td></td>
</tr>
<tr>
<td>b. Conventions</td>
<td></td>
</tr>
<tr>
<td>VI. Reviews and audits</td>
<td></td>
</tr>
<tr>
<td>a. Purpose</td>
<td></td>
</tr>
<tr>
<td>b. Review requirements</td>
<td></td>
</tr>
<tr>
<td>i. Software requirements review</td>
<td></td>
</tr>
<tr>
<td>ii. Design reviews</td>
<td></td>
</tr>
<tr>
<td>iii. Software verification and validation reviews</td>
<td></td>
</tr>
<tr>
<td>iv. Functional audit</td>
<td></td>
</tr>
<tr>
<td>v. Physical audit</td>
<td></td>
</tr>
<tr>
<td>vi. In-process audit</td>
<td></td>
</tr>
<tr>
<td>vii. Management reviews</td>
<td></td>
</tr>
<tr>
<td>VII. Test</td>
<td></td>
</tr>
<tr>
<td>VIII. Problem reporting and corrective action</td>
<td></td>
</tr>
<tr>
<td>IX. Tools, techniques, and methodologies</td>
<td></td>
</tr>
<tr>
<td>X. Code control</td>
<td></td>
</tr>
<tr>
<td>XI. Media control</td>
<td></td>
</tr>
<tr>
<td>XII. Supplier control</td>
<td></td>
</tr>
<tr>
<td>XIII. Records collection, maintenance, and retention</td>
<td></td>
</tr>
<tr>
<td>XIV. Training</td>
<td></td>
</tr>
<tr>
<td>XV. Risk management</td>
<td></td>
</tr>
</tbody>
</table>

relationship with cost in that deciding to meet high-quality standards at the beginning of the project/operation ultimately may reduce maintenance and troubleshooting costs in the long term. This a Design for Six Sigma theme: Avoid design–code–test cycles.

Joseph Juran, one of the world’s leading quality theorists, has been advocating the analysis of quality-related costs since 1951, when he published the first edition of his Quality Control Handbook (Juran & Gryna, 1988). Feigenbaum (1991) made it one of the core ideas underlying the TQM movement. It is a tremendously powerful tool for product quality, including software quality.
Quality cost is the cost associated with preventing, finding, and correcting defective work. The biggest chunk of quality cost is the cost of poor quality (COPQ), a Six Sigma terminology. COPQ consists of those costs that are generated as a result of producing defective software. This cost includes the cost involved in fulfilling the gap between the desired and the actual software quality. It also includes the cost of lost opportunity resulting from the loss of resources used in rectifying the defect. This cost includes all the labor cost, recoding cost, testing costs, and so on, that have been added to the unit up to the point of rejection. COPQ does not include detection and prevention cost.

Quality costs are huge, running at 20% to 40% of sales (Juran & Gryna, 1988). Many of these costs can be reduced significantly or avoided completely. One key function of a Quality Engineer is the reduction of the total cost of quality associated with a product. Software quality cost equals the sum of the prevention costs and the COPQ as defined below (Pressman, 1997):

1. Prevention costs: The costs of activities that specifically are designed to prevent poor quality. Examples of “poor quality” include coding errors, design errors, mistakes in the user manuals, as well as badly documented or unmaintainable complex code. Note that most of the prevention costs does not fit within the testing budget, and the programming, design, and marketing staffs spend this money. Prevention costs include the following:
   a. DFSS team cost
   b. Quality planning
   c. Formal technical reviews
   d. Test equipment
   e. Training

2. Appraisal costs (COPQ element): The are costs of activities that are designed to find quality problems, such as code inspections and any type of testing. Design reviews are part prevention and part appraisal to the degree that one is looking for errors in the proposed software design itself while doing the review and an appraisal. The prevention is possible to the degree that one is looking for ways to strengthen the design. Appraisal cost are activities to gain insight into product condition. Examples include:
   a. In-process and interprocess inspection
   b. Equipment calibration and maintenance
   c. Testing

3. Failure costs (COPQ elements): These costs result from poor quality, such as the cost of fixing bugs and the cost of dealing with customer complaints. Failure costs disappear if no defects appeared before shipping the software product to customers. It includes two types:
   a. Internal failure costs—the cost of detecting errors before shipping the product, which includes the following:
      i. Rework
SOFTWARE QUALITY CONCEPTS

ii. Repair

iii. Failure mode analysis

b. External failure costs—the cost of detecting errors after shipping the product. Examples of external failure costs are:

i. Complaint resolution

ii. Product return and replacement

iii. Help-line support

iv. Warranty work

The costs of finding and repairing a defect in the prevention stage is much less that in the failure stage (Boehm, 1981; Kaplan et al. 1995).

Internal failure costs are failure costs that originate before the company supplies its product to the customer. Along with costs of finding and fixing bugs are many internal failure costs borne outside of software product development. If a bug blocks someone in the company from doing one’s job, the costs of the wasted time, the missed milestones, and the overtime to get back onto schedule are all internal failure costs. For example, if the company sells thousands of copies of the same program, it will probably require printing several thousand copies of a multicolor box that contains and describes the program. It (the company) will often be able to get a much better deal by booking press time with the printer in advance. However, if the artwork does not get to the printer on time, it might have to pay for some or all of that wasted press time anyway, and then it also may have to pay additional printing fees and rush charges to get the printing done on the new schedule. This can be an added expense of many thousands of dollars. Some programming groups treat user interface errors as low priority, leaving them until the end to fix. This can be a mistake. Marketing staff needs pictures of the product’s screen long before the program is finished to get the artwork for the box into the printer on time. User interface bugs—the ones that will be fixed later—can make it hard for these staff members to take (or mock up) accurate screen shots. Delays caused by these minor design flaws, or by bugs that block a packaging staff member from creating or printing special reports, can cause the company to miss its printer deadline. Including costs like lost opportunity and cost of delays in numerical estimates of the total cost of quality can be controversial. Campanella (1990) did not include these in a detailed listing of examples. Juran and Gryna (1988) recommended against including costs like these in the published totals because fallout from the controversy over them can kill the entire quality cost accounting effort. These are found very useful, even if it might not make sense to include them in a balance sheet.

External failure costs are the failure costs that develop after the company supplies the product to the customer, such as customer service costs, or the cost of patching a released product and distributing the patch. External failure costs are huge. It is much cheaper to fix problems before shipping the defective product to customers. The cost rules of thumb are depicted in Figure 1.2. Some of these costs must be treated with care. For example, the cost of public relations (PR) efforts to soften the publicity effects of bugs is probably not a huge percentage of the company’s PR budget. And thus the entire PR budget cannot be charged as a quality-related cost. But any money
that the PR group has to spend to cope specifically with potentially bad publicity because of bugs is a failure cost. COPQ is the sum of appraisal, internal and external quality costs (Kaner, 1996).

Other intangible quality cost elements usually are overlooked in literature (see Figure 1.3). For example, lost customer satisfaction and, therefore, loyalty, lost sales, longer cycle time, and so on. These type of costs can alleviate the total COPQ, which handsomely can be avoided via a thorough top-down DFSS deployment approach. See DFSS deployment chapters for further details (Chapter 8).

1.7 SOFTWARE QUALITY MEASUREMENT

The software market is growing continuously, and users often are dissatisfied with software quality. Satisfaction by users is one of the outcomes of software quality and quality of management.

![Diagram showing measured and not measured quality cost elements.]

FIGURE 1.2 Internal versus external quality cost rules of thumb.

FIGURE 1.3 Measured and not measured quality cost elements.
SOFTWARE QUALITY CONCEPTS

Quality can be defined and measured by its attributes. A proposed way that could be used for measuring software quality factors is given in the following discussion.\(^6\) For every attribute, there is a set of relevant questions. A membership function can be formulated based on the answers to these questions. This membership function can be used to measure the software quality with respect to that particular attribute. It is clear that these measures are fuzzy (subjective) in nature.

The following are the various attributes that can be used to measure software quality:

1.7.1 Understandability

Understandability can be accomplished by requiring all of the design and user documentation to be written clearly. A sample of questions that can be used to measure the software understandability:

- Do the variable names describe the functional property represented? (V1)
- Do functions contain adequate comments? (C1)
- Are deviations from forward logical flow adequately commented? (F1)
- Are all elements of an array functionally related? (A1)
- Are the control flow of the program used adequately? (P1)

The membership function for measuring the software quality with respect to understandability can be defined as follows:

\[
\mu_{\text{Understandability}} = f_1(V1, C1, F1, A1, P1)
\]

1.7.2 Completeness

Completeness can be defined as the presence of all necessary parts of the software system, with each part fully developed. This means that\(^7\) if the code calls a module from an external library, the software system must provide a reference to that library and all required parameters must be passed. A sample of questions that can be used to measure the software completeness:

The membership function for measuring the software quality with respect to completeness can be defined as follows:

\[
\mu_{\text{Completeness}} = f_2(C2, P2, S2, E2)
\]

Are all essential software system components available? (C2)
Does any process fail for lack of resources? (P2)
Does any process fail because of syntactic errors? (S2)
Are all potential pathways through the code accounted for, including proper error handling? (E2)

1.7.3 Conciseness

Conciseness means to minimize the use of redundant information or processing. A sample of questions that can be used to measure the software conciseness:

Is all code reachable? (C3)
Is any code redundant? (R3)
How many statements within loops could be placed outside the loop, thus reducing computation time? (S3)
Are branch decisions too complex? (B3)

The membership function for measuring the software quality with respect to conciseness can be defined as follows:

$$\mu_{Conciseness} = f_3(C3, R3, S3, B3)$$

1.7.4 Portability

Portability can be the ability to run the software system on multiple computer configurations or platforms. A sample of questions that can be used to measure the software portability:

Does the program depend upon system or library routines unique to a particular installation? (L4)
Have machine-dependent statements been flagged and commented? (M4)
Has dependency on internal bit representation of alphanumeric or special characters been avoided? (R4)
How much effort would be required to transfer the program from one hardware/software system or environment to another? (E4)

The membership function for measuring the software quality with respect to portability can be defined as follows:

$$\mu_{Portability} = f_4(L4, M4, R4, E4)$$
1.7.5 Consistency

Consistency means the uniformity in notation, symbols, appearance, and terminology within the software system or application. A sample of questions that can be used to measure the software consistency:

- Is one variable name used to represent different logical or physical entities in the program? (V5)
- Does the program contain only one representation for any given physical or mathematical constant? (P5)
- Are functionally similar arithmetic expressions similarly constructed? (F5)
- Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements? (S5)

The membership function for measuring the software quality with respect to consistency can be defined as follows:

$$\mu_{\text{Consistency}} = f_5(V5, P5, F5, S5)$$

1.7.6 Maintainability

Maintainability is to provide updates to satisfy new requirements. A maintainable software product should be well documented, and it should not be complex. A maintainable software product should have spare capacity of memory storage and processor utilization and other resources. A sample of questions that can be used to measure the software maintainability:

- Has some memory capacity been reserved for future expansion? (M6)
- Is the design cohesive (i.e., does each module have distinct, recognizable functionality)? (C6)
- Does the software allow for a change in data structures? (S6)
- Is the design modular? (D6)
- Was a software process method used in designing the software system? (P6)

The membership function for measuring the software quality with respect to maintainability can be defined as follows:

$$\mu_{\text{Maintainability}} = f_6(M6, C6, S6, D6, P6)$$

1.7.7 Testability

A software product is testable if it supports acceptable criteria and evaluation of performance. For a software product to have this software quality, the design must not be complex. A sample of questions that can be used to measure the software testability:
Are complex structures used in the code? (C7)
Does the detailed design contain clear pseudo-code? (D7)
Is the pseudo-code at a higher level of abstraction than the code? (P7)
If tasking is used in concurrent designs, are schemes available for providing adequate test cases? (T7)

The membership function for measuring the software quality with respect to testability can be defined as follows:

\[ \mu_{\text{Testability}} = f_7(C7, D7, P7, T7) \]

1.7.8 Usability

Usability of a software product is the convenience and practicality of using the product. The easier it is to use the software product, the more usable the product is. The component of the software that influence this attribute the most is the graphical user interface (GUI).\(^8\) A sample of questions that can be used to measure the software usability:

Is a GUI used? (G8)
Is there adequate on-line help? (H8)
Is a user manual provided? (M8)
Are meaningful error messages provided? (E8)

The membership function for measuring the software quality with respect to usability can be defined as follows:

\[ \mu_{\text{Usability}} = f_8(G8, H8, M8, E8) \]

1.7.9 Reliability

Reliability of a software product is the ability to perform its intended functions within a particular environment over a period of time satisfactorily. A sample of questions that can be used to measure the software reliability:

Are loop indexes range-tested? (L9)
Is input data checked for range errors? (I9)
Is divide-by-zero avoided? (D9)
Is exception handling provided? (E9)

SOFTWARE QUALITY CONCEPTS

The membership function for measuring the software quality with respect to reliability can be defined as follows:

$$ \mu_{\text{Reliability}} = f_9(L_9, I_9, D_9, E_9) $$

1.7.10 Structuredness

Structuredness of a software system is the organization of its constituent parts in a definite pattern. A sample of questions that can be used to measure the software structuredness:

- Is a block-structured programming language used? (S10)
- Are modules limited in size? (M10)
- Have the rules for transfer of control between modules been established and followed? (R10)

The membership function for measuring the software quality with respect to structuredness can be defined as follows:

$$ \mu_{\text{Structuredness}} = f_{10}(S_{10}, M_{10}, R_{10}) $$

1.7.11 Efficiency

Efficiency of a software product is the satisfaction of goals of the product without waste of resources. Resources like memory space, processor speed, network bandwidth, time, and so on. A sample of questions that can be used to measure the software efficiency:

- Have functions been optimized for speed? (F11)
- Have repeatedly used blocks of code been formed into subroutines? (R11)
- Has the program been checked for memory leaks or overflow errors? (P11)

The membership function for measuring the software quality with respect to efficiency can be defined as follows:

$$ \mu_{\text{Efficiency}} = f_{11}(F_{11}, R_{11}, P_{11}) $$

1.7.12 Security

Security quality in a software product means the ability of the product to protect data against unauthorized access and the resilience of the product in the face of malicious or inadvertent interference with its operations. A sample of questions that can be used to measure the software security:
Does the software protect itself and its data against unauthorized access and use? (A12)
Does it allow its operator to enforce security policies? (S12)
Are security mechanisms appropriate, adequate, and correctly implemented? (M12)
Can the software withstand attacks that can be anticipated in its intended environment? (W12)
Is the software free of errors that would make it possible to circumvent its security mechanisms? (E12)
Does the architecture limit the potential impact of yet unknown errors? (U12)

The membership function for measuring the software quality with respect to security can be defined as follows:

\[ \mu_{\text{Security}} = f_{12}(A12, S12, M12, W12, E12, U12) \]

There are many perspectives within the field on software quality measurement. Some believe that quantitative measures of software quality are important. Others believe that contexts where quantitative measures are useful are they rare, and so prefer qualitative measures.\(^9\) Many researchers have written in the field of software testing about the difficulty of measuring what we truly want to measure (Pressman, 2005, Crosby, 1979).

In this section, the functions \( f_1 \) through \( f_{12} \) can be linear or nonlinear functions. They can be fuzzy measures. The function \( f_i \) can be a value within the unit interval \( f_i \in [0, 1] \), where \( f_i = 1 \) means that the software quality with respect to the attribute \( i \) is the highest, and \( f_i = 0 \) means that the software quality with respect to the attribute \( i \) is the lowest; otherwise the software quality will be relative to the value of \( f_i \).

1.8 SUMMARY

Quality is essential in all products and systems, and it is more so for software systems because modern computer systems do execute millions of instructions per second, and a simple defect that would occur once in a billion times can occur several times a day.

High-quality software would not only decrease cost but also reduce the production time and increase the company’s competence within the software production world.

Achieving a high quality in software systems demands changing and improving the process. An improved process would include defining the quality goal, measuring the software product quality, understanding the process, adjusting the process, using the adjusted process, measuring the results, comparing the results with the goal, and recycling and continue improving the process until the goal is achieved. It also can be achieved by using DFSS as will be discussed in the following chapters.

SOFTWARE QUALITY CONCEPTS

Many quality standards can be used to achieve high-quality software products. Standards can improve quality by enforcing a process and ensuring that no steps are skipped. The standards can establish allowable tolerance or constraints for levels of items. It can achieve a degree of excellence.

REFERENCES

Jack Campanella, (1990), Principles of Quality Costs, 2nd Ed., ASQC Quality Press, Milwaukee, WI.
Watts, S. Humphrey (1997), Introduction to Personal Software Process, Addison Wesley, Boston, MA.