CHAPTER 10

SOFTWARE QUALITY

ACRONYMS

CMMI	Capability Maturity Model Integrated
CoSQ	Cost of Software Quality
COTS	Commercial Off-the-Shelf Software
PDCA	Plan, Do, Check, Act
SQA	Software Quality Assurance
SQM	Software Quality Management
TQM	Total Quality Management
V&V	Verification and Validation

INTRODUCTION

What is software quality and why is it so important that it be pervasive in the SWEBOK Guide? Over the years, authors and organizations have defined the term *quality* differently. To Phil Crosby [1], it was "conformance to user requirements." Watts Humphrey [2] refers to it as "achieving excellent levels of fitness for use." Meanwhile, IBM coined the phrase "market-driven quality," where the "customer is the final arbiter" [3*, p. 8]. More recently, quality has been defined in [4] as "the degree to which a set of inherent characteristics fulfills requirements."

This chapter deals with software quality considerations that transcend the life-cycle processes. Software quality is a ubiquitous concern in software engineering, and so it is also considered in many of the KAs. In summary, the SWEBOK Guide describes a number of ways of achieving software quality. In particular, this KA will cover *static techniques*, those which do not require the execution of the software being evaluated; *dynamic techniques* are covered in the Software Testing KA.

According to [5], a document is a "uniquely identified unit of information for human use." So, throughout this chapter, the term *document* can be taken to include many types of artifacts—notably, a software requirements specification, a user manual, a help file, a software design description, a software test plan, and a programming language listing (source code). In this Knowledge Area, the term *software* unless noted otherwise, means the executable computer program. This chapter discusses quality characteristics of documents and executables.

BREAKDOWN OF SOFTWARE QUALITY TOPICS

The breakdown of the Software Quality KA is presented below, together with brief descriptions of the topics associated with it. Appropriate references are also given for each of the topics. Figure 1 gives a graphical representation of the top-level decomposition of the breakdown for this KA.

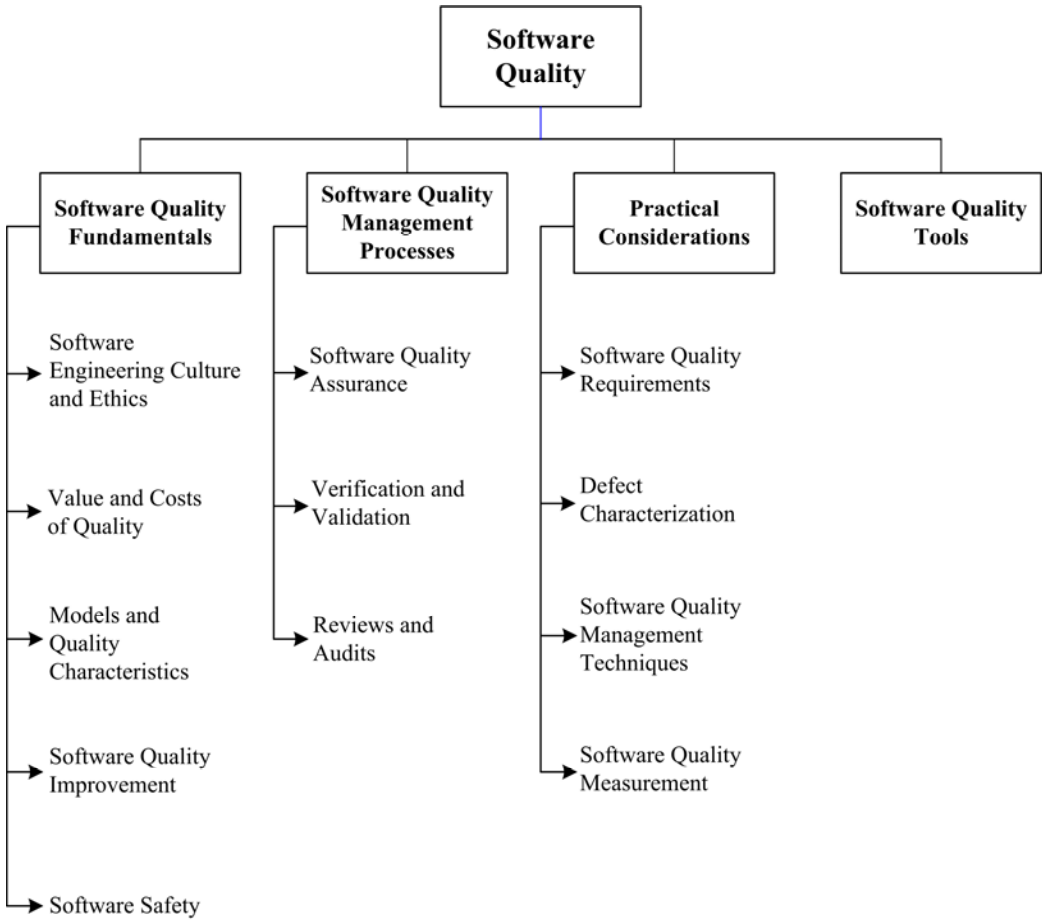

Figure 1. Breakdown of topics for the Software Quality KA

1. Software Quality Fundamentals

Agreement on quality requirements, as well as clear communication to the software engineer on what constitutes quality, require that the many aspects of quality be formally defined and discussed.

A software engineer should understand the underlying meanings of quality concepts and characteristics and their value to the software under development or to maintenance. The important concept is that the software requirements define the required quality attributes of the software. Software requirements influence the measurement methods and acceptance criteria for assessing the degree to which the software and related documentation achieve the desired quality levels.

1.1. Software Engineering Culture and Ethics

[6*, c2s3.5] [3*, c1s4]

Software engineers are expected to share a commitment to software quality as part of their culture. A healthy software engineering culture includes many characteristics, not the least of which is the importance of adhering to quality procedures in spite of other priorities.

Ethics also plays a significant role in software quality, the culture, and the attitudes of software engineers. The IEEE Computer Society and the ACM have developed a code of ethics and professional practice (see Codes of Ethics and Professional Conduct in the Software Engineering Professional Practice KA).

1.2. Value and Costs of Quality

[7*, c17, c22]

Defining and then achieving the quality of software and related documentation is not simple. For any engineered product, there are many desired qualities relevant to a particular perspective of the product, to be discussed and determined at the time that the product requirements are set down. Quality characteristics may or may not be required, or they may be required to a greater or lesser degree, and tradeoffs may be made among them.

One method of determining the level of quality in a software product is Cost of Software Quality (CoSQ). This method proposes that the more an organization spends on activities related to poor software quality, the lower the level of software product quality. There are two cost of quality categories: costs to control software quality (prevention costs and appraisal costs) and costs of poor software quality (internal failure costs and external failure costs). Prevention costs are usually not specific to a project but extend across an organization and include investments in improvement efforts, quality infrastructure, training, audits, and management reviews. Appraisal costs are usually specific to a project and include reviews and testing. Costs of internal and external failures include activities to respond to software problems discovered prior to releasing the software to the customer and after release to the customer, respectively.

A motivation behind a software project is the desire to create software that provides value to stakeholders, and this value may or may not be quantified as a cost. The customer will have some maximum cost in mind, in return for which it is expected that the basic purpose of the software will be fulfilled. The customer may also have some expectation as to the quality of the software. Sometimes customers may not have thought through the quality issues or their related costs. Is any given characteristic merely decorative or is it essential to the software? If the answer lies somewhere in between, as is almost always the case, it is a matter of making the customer a part of the decision process and fully aware of both costs and benefits. Ideally, most of these decisions will be made in the software requirements process (see the Software Requirements KA), but these issues may arise throughout the software life cycle. There is no definite rule as to how these decisions should be made, but the software engineer should be able to present quality alternatives and their costs.

1.3. Models and Quality Characteristics

[3*, c24s1] [7*, c2s4] [8*, c17]

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. [9] defines two related models of quality (product quality and quality in use).

1.3.1. Software engineering process quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria that evaluate the capabilities of software organizations are primarily project organization and management considerations and, as such, are covered in the Software Engineering Management and Software Engineering Process KAs.

Of course, it is not possible to completely distinguish the quality of the process from the quality of the product, because the outcomes of processes are the products. Determining whether a process will consistently produce products of desired quality is not simple.

The software engineering process, discussed in the Software Engineering Process KA of this Guide, influences the quality characteristics of software products, which in turn affect quality-in-use as perceived by the customer.

1.3.2. Software product quality

The software engineer needs, first of all, to determine the real purpose of the software. In this regard, the customer's requirements come first and they include quality requirements—not just functional requirements. Thus, the software engineer has a responsibility to elicit quality requirements that may not be explicit at the outset and to discuss their importance as well as the level of difficulty in attaining them. All processes associated with software quality (for example, building, checking, and improving quality) will be designed with these requirements in mind and will carry additional costs.

The meaning of the term *product* is extended to include any artifact that is the output of any process used to build the final software product. Examples of a product include, but are not limited to, an entire system requirements specification, a software requirements specification for a software component of a system, a software design description, source code, software test documentation, or reports produced as a result of quality analysis tasks. While some treatments of quality are described in terms of the final software and system performance, sound engineering practice requires that intermediate products relevant to quality be evaluated throughout the software engineering process.

1.4. Software Quality Improvement

[3*, c1s4, 10*, c24, 11*, c11s2.4]

The quality of software products can be improved through an iterative process of continual improvement, which requires management control, coordination, and feedback from many concurrent processes: (1) the software life-cycle processes, (2) the process of fault/defect detection, removal, and prevention, and (3) the quality-improvement process.

The theory and concepts behind quality improvement—such as *building in quality* through the prevention and early detection of defects, continual improvement, and customer focus—are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it. Approaches such as the Deming improvement cycle of *Plan, Do, Check*, and *Act* (PDCA), evolutionary delivery, kaizen, and Quality Function Deployment (QFD) offer techniques by which quality objectives can be met.

Management sponsorship supports process and product evaluations and the resulting findings. Then an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship must be solicited frequently by implementing proactive communication activities.

1.5. Software Safety

[10*, c11s3]

Safety-critical systems are those in which a system failure could harm human life, other living things, physical structures, or the environment. The software in these software-intensive (also known as "software-reliant") systems is safety-critical. There are increasing numbers of applications of safety-critical software. Examples of systems with safety-critical software include mass transit systems, chemical manufacturing plants, and medical devices. The failure of software in these systems could have catastrophic effects. There are industry standards, such as DO-178C [12], and emerging processes, tools, and techniques for developing safety-critical software. The intent of these standards, tools, and techniques is to reduce the risk of injecting faults into the software and thus reducing the probability of software failures.

Safety-critical software can be categorized as direct or indirect. Direct is that software embedded in a safety-critical system, such as the flight control computer of an aircraft. Indirect includes software applications used to develop safety-critical software. Indirect software are included in software engineering environments and software test environments.

Three complimentary techniques for reducing the risk of failure are avoidance, detection and removal, and damage limitation. These techniques impact software functional requirements, software performance requirements, and development processes. Increasing levels of risk imply increasing levels of software quality assurance and control techniques, such as inspections. Higher risk levels may, for example, imply more thorough inspections of requirements, design, and code or the use of more formal analytical techniques.

2. Software Quality Management Processes

Software quality management (SQM) applies to all perspectives of software engineering processes, products, and resources. SQM defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels.

SQM can notably be decomposed into three sub-categories: software quality planning, software quality assurance (SQA), and software quality control (SQC). Software quality planning includes determining which quality standards are to be used, defining specific quality goals, and estimating the effort and schedule of software quality activities. In some cases, software quality planning also includes activities for defining the software quality processes to be used. Software quality assurance includes auditing software development processes to provide confidence that the software products will meet quality requirements. SQA is generally concerned with auditing processes. Software quality control activities examine specific project artifacts (documents and executables) to determine whether they comply with quality requirements. SQC evaluates intermediate products as well as the final products. Activities for improving software (process and product) quality can be included in any of the above three categories or, in an increasing number of organizations over the past years, into a separate category: Software Process Improvement (SPI), which often spans across many projects within an organization (see the Software Engineering Process KA).

SQM processes consist of tasks and techniques to indicate how software plans (for example, management, development, or software-configuration management plans) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

As described in this KA, SQM processes are closely related; they can overlap and are sometimes even combined. They seem largely reactive in nature because they address the processes as practiced and the products as produced; however, they have a major role at the planning stage in being proactive in terms of the processes and procedures needed to meet the quality requirements needed by the stakeholders in the software.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life-cycle processes can increase the potential for producing a quality product (see the Software Engineering Management KA for related material on risk management).

2.1. Software Quality Assurance

[7*, c4-c6, c11, c12, c26-27]

Software quality assurance is not testing. SQA processes provide assurance that the software products and processes in the life cycle conform to their specified requirements by planning and enacting a set of activities to provide adequate confidence that quality is being built into the software. This means ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. SQA encompasses activities across the entire life cycle. SQA seeks to maintain quality throughout the development and maintenance of the product by the execution of a variety of activities at each stage, which can result in early identification of problems (an almost inevitable feature of any complex activity). The role of SQA with respect to process is to ensure that planned processes are appropriate and later implemented according to plan and that relevant measurement processes are provided to the appropriate organization.

The Software Quality Plan (in some industry sectors it is termed the Software Quality Assurance Plan) defines the means that will be used to ensure that software developed for a specific product satisfies the user's requirements and is of the highest quality possible within project constraints. In order to do so, it must first ensure that the quality target is clearly defined and understood.

The specific quality activities and tasks are laid out—with their costs and resource requirements, their overall management objectives, and their schedule in relation to those objectives—in the software engineering management, software development, or software maintenance plans. The SQA plan should be consistent with the software configuration management plan (refer to the Software Configuration Management KA). The SQA plan identifies documents, standards, practices, and conventions governing the project and how they will be checked and monitored to ensure adequacy and compliance. The SQA plan also identifies measures; statistical techniques; procedures for problem reporting and corrective action; resources such as tools, techniques, and methodologies; security for physical media; training; and SQA reporting and documentation. Moreover, the SQA plan addresses the software quality assurance activities of any other type of activity described in the software plans—such as procurement of supplier software to the project, commercial off-the-shelf software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities that are critical to software quality.

2.2. Verification & Validation

[10*, c2s2.3, c21s3.3, c8, c15s1.1]

For purposes of brevity, verification and validation (V&V) are treated as a single topic in this Guide. As stated in [13], "The purpose of V&V is to help the development organization build quality into the system during the life cycle. V&V processes provide an objective assessment of products and processes throughout the life cycle. This assessment demonstrates whether the requirements are correct, complete, accurate, consistent, and testable. The V&V processes determine whether the development products of a given activity conform to the requirements of that activity and whether the product satisfies its intended use and user needs."

Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built—that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation to both the product's immediate predecessor and the specifications it must meet.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents describe the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity will help in the careful planning of the techniques and resources needed to fulfill their purposes. The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

2.3. Reviews and Audits

[10*, c24s3] [14*]

For purposes of brevity, reviews and audits are described within a single topic in this Guide. Reviews and audit processes are broadly defined as the static examination of software engineering artifacts with respect to some standards that have been established by the organization or project for those artifacts. Reviews and audits are distinguished by their purpose and by the type of artifact being reviewed or audited:

- Management reviews
- Technical reviews
- Inspections
- Walk-throughs
- Audits

2.3.1. Management reviews

"The purpose of a management review is to monitor progress, determine the status of plans and schedules, or evaluate the effectiveness of management approaches used to achieve fitness for purpose. Management reviews support decisions about corrective actions, changes in the allocation of resources, or changes to the scope of the project" [14*]. Management reviews determine the adequacy of plans, schedules, and requirements to achieve project objectives; they also evaluate actual progress with respect to plans and schedules, and determine inconsistencies between requirements and actual project results. These reviews may be performed on products such as audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others. (Refer to the Software Engineering Management and the Software Configuration Management KAs for related material.)

2.3.2. Technical reviews

"The purpose of a technical review is to evaluate a software product by a team of qualified personnel to determine its suitability for its intended use and identify discrepancies from specifications and standards. It provides management with evidence to confirm the technical status of the project" [14*].

Specific roles must be established in a technical review: a decision maker, a review leader, a recorder, and technical staff to support the review activities. A technical review requires that mandatory inputs be in place in order to proceed:

- Statement of objectives
- Specific software product
- Specific project management plan
- Issues list associated with this product
- Technical review procedure

The team follows the review procedure. A technically qualified individual presents an overview of the product, and the examination is conducted during one or more meetings.

The technical review is completed once all the activities listed in the examination have been completed.

2.3.3. Inspections

"The purpose of an inspection is to detect and identify software product anomalies" [14*]. Two important differentiators of inspections as opposed to reviews are as follows:

1. An individual holding a management position over any member of the inspection team shall not participate in the inspection.
2. An inspection is to be led by an impartial facilitator who is trained in inspection techniques.

Software inspections always involve the author of an intermediate or final product; other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (two to five) inspectors. The members of an inspection team may possess different expertise, such as domain expertise, software design method expertise, or programming language expertise. Inspections are usually conducted on one relatively small section of the product at a time. Each team member must examine the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique (refer to section 3.3.3) to a small section of the product or to the entire product with a focus only on one aspect—for example, interfaces. Any anomaly found is documented and sent to the inspection leader. During the inspection, the inspection leader conducts the session and verifies that everyone has prepared for the inspection. A checklist, with anomalies and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies (see section 3.2 "Defect Characterization") and is reviewed for completeness and accuracy by the team.

The inspection exit decision must correspond to one of the following three criteria:

1. Accept with no or, at most, minor reworking
2. Accept with rework verification
3. Re-inspect

Inspection meetings typically last a few hours, whereas technical reviews and audits are usually broader in scope and take longer.

2.3.4. Walk-throughs

"The purpose of a systematic walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product. The major objectives are to:

- Find anomalies
- Improve the software product
- Consider alternative implementations
- Evaluate conformance to standards and specifications
- Evaluate the usability and accessibility of the software product" [14*]

The walk-through is similar to an inspection but is typically conducted less formally. The walk-through is primarily organized by the software engineer to give his teammates the opportunity to review his work as an assurance technique.

2.3.5. Audits

"The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures" [14*]. The audit is a formally organized activity with participants having specific roles—such as lead auditor, another auditor, a recorder, or an initiator—and including a representative of the audited organization. The audit will identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits, such as those identified in the standard [14*], the important point is that they can occur on almost any product at any stage of the development or maintenance process.

3. Practical Considerations

3.1. Software Quality Requirements

[10*, c11s1, 15*, c15s3.2.2,c15s3.3.1, c16s9.10, 16*, c12]

3.1.1. Influence factors

Various factors influence planning, management, and selection of SQM activities and techniques, including:

- The domain of the system in which the software will reside (safety-critical, mission-critical, business-critical, security-critical)
- System and software functional and non-functional requirements
- The commercial (external) or standard (internal) components to be used in the system
- The specific software engineering standards applicable
- The methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- The budget, staff, project organization, plans, and scheduling of all the processes
- The intended users and use of the system
- The integrity level of the system

Information on these factors influence how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which of those resources will impose bounds on the efforts.

3.1.2. Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human or operational) is the main quality requirement over and above basic functionality. This is the case for the following reasons: system failures affect a large number of people; users often reject systems that are unreliable, unsafe, or insecure; system failure costs may be enormous; and undependable systems may cause information loss. System and software dependability include such characteristics as availability, reliability, safety, and security. When developing dependable software, you must notably apply special care to ensure that faults are not injected into the intermediate deliverables or the final software product and that verification, validation, and testing processes, techniques, methods, and tools identify faults that impact dependability as early as possible in the life cycle. Additionally, mechanisms may need to be in place in the software to guard against external attacks and to tolerate faults.

3.1.3. Integrity levels of software

The integrity level is determined based on the possible consequences of failure of the software and the probability of failure. For software in which safety or security is important, techniques such as hazard analysis for safety or threat analysis for security may be used to develop a software safety plan that would identify where potential trouble spots lie. The failure history of similar software may also help identify which techniques will be most useful in detecting faults and assessing quality. As noted in [15*], “the integrity levels can be applied during development to allocate additional verification and validation efforts to high-integrity components.”

3.2. *Defect Characterization*

[3*, c3s3, c8s8, c10s2]

SQM processes find defects. Characterizing those defects leads to an understanding of the product, facilitates corrections to the process or the product, and informs management or the customer of the status of the process or product. Many defect (fault) taxonomies exist and, while attempts have been made to gain consensus on a fault and failure taxonomy, the literature indicates that there are quite a few in use. Defect (anomaly) characterization is also used in audits and reviews, with the review leader often presenting a list of anomalies provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, is not really of any use in identifying the underlying causes of the defects since specific types of problems need to be grouped together in order for determinations to be made about them. The point is to establish a defect taxonomy that is meaningful to the organization and to the software engineers.

SQM discovers information at all stages of software development and maintenance. In some cases, the word *defect* refers to a *fault* as defined below. However, different cultures and standards may use somewhat different meanings for these terms, which has led to attempts to define them. Partial definitions taken from [5] are:

- *Error*: "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition."
- *Error*: A slip or mistake that a person makes. "A human action that produces an incorrect result."
- *Defect*: An "imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced." A defect is caused by a person committing an error.
- *Fault*: An "incorrect step, process, or data definition in computer program." The encodement of a human error in source code. *Fault* is the formal name of what is often commonly termed a *bug.*
- *Failure*: An "event in which a system or system component does not perform a required function within specified limits." A failure "may be produced when a fault is encountered."

Failures found in testing as a result of software faults are included as defects in the discussion in this section. Reliability models are built from failure data collected during software testing or from software in service, and thus can be used to predict future failures and to assist in decisions on when to stop testing.

One probable action resulting from SQM findings is to remove the defects from the product under examination. Other actions attempt to eliminate or reduce the causes of the defects. These actions include analyzing and summarizing the findings in order to find root causes, and using measurement techniques to improve the product and the process as well as to track the defects and their removal. Process improvement is primarily discussed in the Software Engineering Process KA, with the SQM process being a source of information.

Data on inadequacies and defects found by SQM techniques may be lost unless they are recorded. For some techniques (for example, technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used (see section 4 "Software Quality Tools"), the tool output may provide the defect information. Reports about defects are provided to the management of the organization.

3.3. *Software Quality Management Techniques*

[10*, c15s1, p417, c12s5] [14*] [8*, c17] [7*, c7s3]

SQM techniques can be categorized in many ways but a straightforward approach uses just two categories: static and dynamic. Dynamic techniques involve executing the software; static techniques involve analyzing documents and source code but not executing the software. Within the static category are people-intensive techniques and analytical techniques. Dynamic techniques include modeling and testing.

3.3.1. Static techniques

Static techniques involve examination of the project documentation and software source code as well as other information about the software products, without executing them. These techniques may include people-intensive activities (as defined in 3.3.1.1) or analytical activities (as defined in 3.3.1.2) conducted by individuals, with or without the assistance of automated tools.

3.3.1.1 People-intensive techniques

The setting for people-intensive techniques, including reviews and audits (see section 2.3 "Reviews and Audits"), may vary from a formal meeting to an informal gathering or a desk-check situation, but (usually, at least) two or more people are involved. Preparation ahead of time may be necessary. Resources other than the items under examination may include checklists and results from analytical techniques and testing.

3.3.1.2 Analytical techniques

A software engineer generally applies analytical techniques. Sometimes several software engineers use the same technique, each applying it to different parts of the product. Some techniques are tool-driven; others are manual. Some may find defects directly, but they are typically used to support other techniques. Some also include various assessments as part of overall quality analysis. Examples of such techniques include complexity analysis, control flow analysis, and algorithmic analysis.

Each type of analysis has a specific purpose and not all types are applied to every project. For example, complexity analysis is useful for determining whether or not the design or implementation is too complex to develop correctly, test, or maintain. The results of a complexity analysis may also be used in developing test cases. Defect-finding techniques, such as control flow analysis, may also be used to support another activity. For software with many algorithms, algorithmic analysis is important, especially when an incorrect algorithm could cause a catastrophic result.

Other, more formal, types of analytical techniques are known as *formal methods.* They are notably used to verify software requirements and designs. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements. (See also "Formal Methods" in the Software Engineering Models and Methods KA)

3.3.2. Dynamic techniques

Dynamic means "executing the software." Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation and model analysis may be considered dynamic (see the Software Engineering Models and Methods KA). Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. In this sense, code reading may also qualify as a dynamic technique. This discrepancy in categorizing indicates that people with different roles in the organization may consider and apply these techniques differently.

Some testing may thus be performed in the development process, SQA process, or V&V process—again, depending on project organization. Because SQM plans address testing, this section includes some comments on testing even though the Software Testing KA is devoted entirely to this subject.

3.3.3. Testing

Two types of testing may fall under the headings SQA and V&V, because of their responsibility for the quality of the materials used in the project:

- Evaluation and tests of tools to be used on the project
- Conformance tests (or review of conformance tests) of components and COTS products to be used in the product

Sometimes an independent V&V organization may be asked to monitor the test process or witness the actual execution to ensure that it is conducted in accordance with specified procedures. Again, V&V may be called upon to evaluate the testing itself: adequacy of plans and procedures, and adequacy and accuracy of results.

Another type of testing that may fall under the heading of V&V organization is third-party testing. The third party is not the developer nor is in any way associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements.

3.4. Software Quality Measurement

[10*, p90] [8*, c17] [3*, c4]

Software quality measurements are used to support decision making. With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

Decisions supported by software quality measurement include determining levels of software quality (notably because models of software product quality include measures to determine the degree to which the software product achieves quality goals); managerial questions about effort, cost, and schedule; determining when to stop testing and release a product (see also the sub-topic “Termination”, under topic 5.1 “Practical Considerations” in the Software Testing KA) ; and determining the efficacy of process improvement efforts.

The cost of SQM processes is an issue frequently raised in deciding how a project or a software development and maintenance group should be organized. Often, generic models of cost are used, which are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in the development process. Software quality measurement data collected internally may give a better picture of cost within this project or organization.

While the software quality measurement data may be useful in itself (for example, the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures. These include:

- Statistically based (for example, Pareto analysis, run charts, scatter plots, normal distribution)
- Statistical tests (for example, the binomial test, chi-squared test)
- Trend analysis
- Prediction (for example, reliability models)

The statistically based techniques and tests often provide a snapshot of the more troublesome areas of the software product under examination. The resulting charts and graphs are visualization aids, which the decision makers can use to focus resources where they appear most needed. Results from trend analysis may indicate that a schedule is being met, such as in testing, or that certain classes of faults will become more likely to occur unless some corrective action is taken in development. The predictive techniques assist in estimating testing effort and schedule and in predicting failures. (More discussion on measurement in general appears in the Software Engineering Process and Software Engineering Management KAs. More specific information on testing measurement is presented in the Software Testing KA.)

Defect analysis includes measuring defect occurrences, applying statistical methods to understand the types of defects that occur most frequently, and determining methods to reduce or eliminate their recurrence. They also aid in understanding the trends, how well detection techniques are working, and how well the development and maintenance processes are progressing. From these measurement methods, defect profiles can be developed for a specific application domain. Then, for the next software project within that organization, the profiles can be used to guide the SQM processes—that is, to expend the effort where problems are most likely to occur. Similarly, benchmarks, or defect counts typical of that domain, may serve as one aid in determining when the product is ready for delivery. (Discussion on using data from SQM to improve development and maintenance processes appears in the Software Engineering Management and Software Engineering Process KAs.)

4. Software Quality Tools

Categories of software quality tools include:

- Static analysis tools that input source code, perform syntactical and semantic analysis, measure different attributes— for example, looking for potential memory leaks—and present results to users. There is a large variety in the depth, thoroughness, and scope of static analysis tools.
- Dynamic analysis tools often include three functions. One function is to embed flags, variables, and functions—collectively, these are sometimes termed *software monitors*—into the source code. During runtime, the software monitors extract information from the executable and uploads it, via an interface, to the second function. The second function executes separately from the software under test and collects, collates, and stores data from the runtime software monitors. A third function of these types of tools is to analyze the results and present them to users. The types of data collected include branches and paths executed by testing— generally termed *code coverage*—and values of data items at different stages of execution.
- Tools that facilitate and partially automate reviews and inspections of documents and code. This category of tool does not involve executing the software under development. These types of tools include workflow to route work to different participants in order to partially automate and control a review process. These tools allow users to enter defects found during inspections and reviews for later removal.
- There are tools that help organizations perform software safety hazard analysis. These tools, for example, automate failure mode and effects analysis (FMEA) and fault tree analysis (FTA).
- Tools that manage and track software problems. Anomalies discovered

during software testing are entered into the tool for subsequent analysis, disposition, and resolution. These tools sometimes include the ability to design workflows to automate the resolution workflow and for tracking the status of problem resolution.

- Tools that analyze data captured from software engineering environments and software test environments and produce visual displays of quantified data in the form of graphs, charts, and tables. These tools can include the functionality to perform statistical analysis on data sets (for the purpose of discerning trends and making forecasts).

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	[6*] Bott et al., 2000	[3*] Kan, 2002	[7*] Galin, 2004	[14*] IEEE Std 1028-2008	[15*] Moore, 2006	[8*] Naik and Tripathy, 2008	[10*] Sommerville, 2011	[11*] Voland, 2003	[16*] Wieger, 2003
1.Software Quality Fundamentals									
1.1 Software Engineering Culture and Ethics	*c2s3.5*	*c1s4*							
1.2 Value and Cost of Quality			*c17, c22*						
1.3 Models and Quality Characteristics		*c24s1*	*c2s4*			*c17*			
1.4 Software Quality Improvement		*c1s4*					*c24*	*c11s2.4*	
1.5 Software Safety							*c11s3*		
2. Software Quality Management Processes									
2.1 Software Quality Assurance			*c4-c6, c11, c26-27*						
2.2 Verification and Validation							*c2s2.3, c21s3.3, c8, c15s1.1*		
2.3 Reviews and Audits				*			*c24s3*		
3.Software Quality Practical Considerations									
3.1 Software Quality					c15s3.2.2,c15s3.		c11s1		c12

	[6*] Bott et al., 2000	[3*] Kan, 2002	[7*] Galin, 2004	[14*] IEEE Std 1028-2008	[15*] Moore, 2006	[8*] Naik and Tripathy, 2008	[10*] Sommerville, 2011	[11*] Voland, 2003	[16*] Wieger, 2003
Requirements					3.1, c16s9.10				
3.2 Defect Characterization		*c3s3, c8s8, c10s2*							
3.3 SQM Techniques			*c7s3*	*		*c17*	*c15s1, p. 417, c12s5*		
3.4Software Quality Measurement		*c4*				*c17*	*p. 90*		
4. Software Quality Tools									

APPENDIX A. LIST OF FURTHER READINGS

To be completed

[1] P. B. Crosby, *Quality Is Free*: McGraw-Hill, 1979.

[2] W. Humphrey, *Managing the Software Process*: Addison-Wesley, 1989.

[3*] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Boston: Addison-Wesley, 2002.

[4] ISO, "ISO 9000:2005 Quality Management Systems - Fundamentals and Vocabulary," ed: ISO, 2005.

[5] IEEE/ISO/IEC, "IEEE/ISO/IEC 24765: Systems and Software Engineering - Vocabulary," 1st ed, 2010.

[6*] F. Bott*, et al.*, *Professional Issues in Software Engineering*, 3rd ed. New York: Taylor & Francis, 2000.

[7*] D. Galin, *Software Quality Assurance : From Theory to Implementation*: Pearson Education Limited, 2004.

[8*] S. Naik and P. Tripathy, "Software Testing and Quality Assurance: Theory and Practice," ed: Wiley, 2008, p. 648.

[9] ISO/IEC, "ISO/IEC 25010: 2011 Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Systems and Software Quality Models," ed, 2011.

[10*] I. Sommerville, *Software Engineering*, 9th ed. New York: Addison-Wesley, 2011.

[11*] G. Voland, *Engineering by Design*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2003.

[12] RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," ed, 2011.

[13] IEEE, "IEEE Std 1012:2012 for System and Software Verification and Validation," ed, 2012.

[14*] IEEE, "IEEE Std. 1028-2008, Software Reviews and Audits," ed, 2008.

[15*] J. W. Moore, *The Road Map to Software Engineering: A Standards-Based Guide*, 1st ed. Hoboken, NJ: Wiley-IEEE Computer Society Press, 2006.

[16*] K. E. Wiegers, *Software Requirements*, 2nd ed. Redmond, WA: Microsoft Press, 2003.