CHAPTER 10

SOFTWARE QUALITY

ACRONYMS

CMMI	Capability Maturity Model Integration
CoSQ	Cost of Software Quality
COTS	Commercial Off-the-Shelf Software
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
PDCA	Plan-Do-Check-Act
PDSA	Plan-Do-Study-Act
QFD	Quality Function Deployment
SPI	Software Process Improvement
SQA	Software Quality Assurance
SQC	Software Quality Control
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 is included in many knowledge areas (KAs) of the *SWEBOK Guide*?

One reason is that the term *software quality* is overloaded. Software quality may refer: to desirable characteristics of software products, to the extent to which a particular software product possess those characteristics, and to processes, tools, and techniques used to achieve those characteristics. Over the years, authors and organizations have defined the term quality differently. To Phil Crosby, it was "conformance to requirements" [1]. Watts Humphrey refers to it as "achieving excellent levels of "fitness for use" [2]. Meanwhile, IBM coined the phrase "market-driven quality," where the "customer is the final arbiter" [3*, p8].

More recently, software quality is defined as the "capability of software product to satisfy stated and implied needs under specified conditions" [4] and as "the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations" [5]. Both definitions embrace the premise of conformance to requirements. Neither refers to types of requirements (e.g., functional, reliability, performance, dependability, or any other characteristic). Significantly, however, these definitions emphasize that quality is dependent upon requirements.

These definitions also illustrate another reason for the prevalence of software quality throughout this *Guide*: a frequent ambiguity of *software quality* versus *software quality requirements* ("the *-ilities*" is a common shorthand). Software quality requirements are actually attributes of (or constraints on) functional requirements (what the system does). Software requirements may also specify resource usage, a communication protocol, or many other characteristics. This KA attempts clarity by using *software quality* in the broadest sense from the definitions above and by using *software quality requirements* as constraints on functional requirements. Software quality is achieved by conformance to all requirements regardless of what characteristic is specified or how requirements are grouped or named.

Software quality is also considered in many of the SWEBOK KAs because it is a basic parameter of a software engineering effort. For all engineered products, the primary goal is delivering maximum stakeholder value, while balancing the constraints of development cost and schedule; this is sometimes characterized as "fitness for

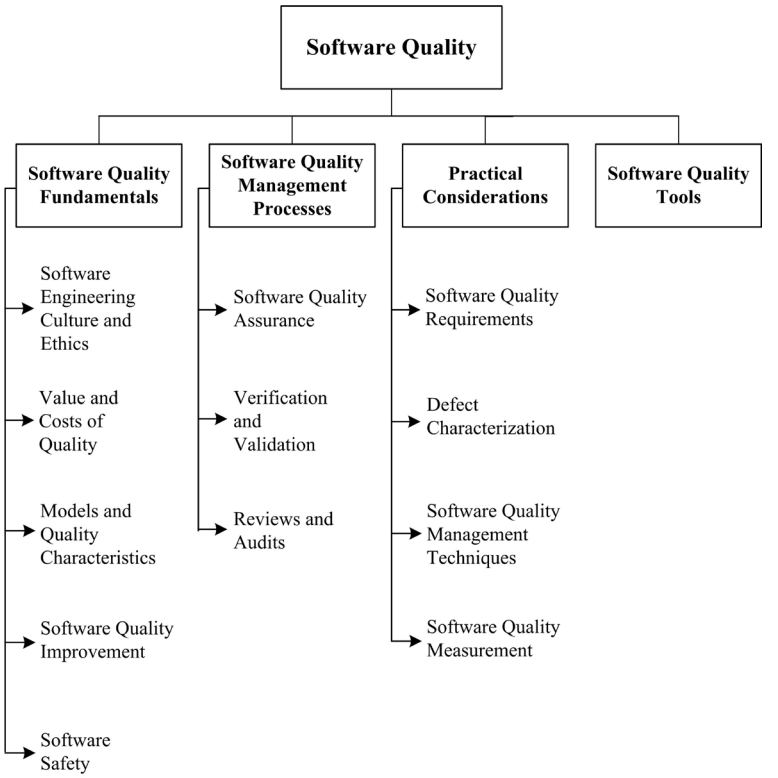

Figure 10.1. Breakdown of Topics for the Software Quality KA

use." Stakeholder value is expressed in requirements. For software products, stakeholders could value price (what they pay for the product), lead time (how fast they get the product), and software quality.

This KA addresses definitions and provides an overview of practices, tools, and techniques for defining software quality and for appraising the state of software quality during development, maintenance, and deployment. Cited references provide additional details.

BREAKDOWN OF TOPICS FOR SOFTWARE QUALITY

The breakdown of topics for the Software Quality KA is presented in Figure 10.1.

1. Software Quality Fundamentals

Reaching agreement on what constitutes quality for all stakeholders and clearly communicating that agreement to software engineers require that the many aspects of quality be formally defined and discussed.

A software engineer should understand quality concepts, characteristics, values, and their application to the software under development or 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

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

Software engineers are expected to share a commitment to software quality as part of their culture. A healthy software engineering culture includes many characteristics, including the understanding that tradeoffs among cost, schedule, and quality are a basic tenant of the engineering of any product. A strong software engineering ethic assumes

that engineers accurately report information, conditions, and outcomes related to quality.

Ethics also play 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 software quality is not simple. 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. To help determine the level of software quality, i.e., achieving stakeholder value, this section presents cost of software quality (CoSQ): a set of measurements derived from the economic assessment of software quality development and maintenance processes. The CoSQ measurements are examples of process measurements that may be used to infer characteristics of a product.

The premise underlying the CoSQ is that the level of quality in a software product can be inferred from the cost of activities related to dealing with the consequences of poor quality. Poor quality means that the software product does not fully "satisfy stated and implied needs" or "established requirements." There are four cost of quality categories: prevention, appraisal, internal failure, and external failure.

Prevention costs include investments in software process improvement efforts, quality infrastructure, quality tools, training, audits, and management reviews. These costs are usually not specific to a project; they span the organization. Appraisal costs arise from project activities that find defects. These appraisal activities can be categorized into costs of reviews (design, peer) and costs of testing (software unit testing, software integration, system level testing, acceptance testing); appraisal costs would be extended to subcontracted software suppliers. Costs of internal failures are those that are incurred to fix defects found during appraisal activities and discovered prior to delivery of the software product to the customer. External failure costs include activities to respond to software problems discovered after delivery to the customer.

Software engineers should be able to use CoSQ methods to ascertain levels of software quality and should also be able to present quality alternatives and their costs so that tradeoffs between cost, schedule, and delivery of stakeholder value can be made.

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. ISO/IEC 25010: 2011 [4] defines product quality and quality in use as two related quality models. Appendix B in the *SWEBOK Guide* provides a list of applicable standards for each KA. Standards for this KA cover various ways of characterizing software quality.

1.3.1. Software 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.

It is not possible to completely distinguish process quality from product quality because process outcomes include products. Determining whether a process has the capability to consistently produce products of desired quality is not simple.

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

1.3.2. Software Product Quality

The software engineer, first of all, must determine the real purpose of the software. In this regard, stakeholder requirements are paramount, and they include quality requirements in addition to functional requirements. Thus, software engineers have a responsibility to elicit quality requirements that may not be explicit at the outset and to understand their importance as well as the level of difficulty in attaining them. All software development processes (e.g., eliciting requirements, designing, constructing, building, checking, improving quality) are designed with these quality requirements in mind and may carry additional development costs if attributes such as safety, security, and dependability are important. The additional development costs help ensure that quality obtained can be traded off against the anticipated benefits.

The term work-product means any artifact that is the outcome of a process used to create the final software product. Examples of a work-product include a system/subsystem specification, a software requirements specification for a software component of a system, a software design description, source code, software test documentation, or reports. While some treatments of quality are described in terms of final software and system performance, sound engineering practice requires that intermediate work-products relevant to quality be evaluated throughout the software engineering process.

1.4. Software Quality Improvement

[3*, c1s4] [9*, c24] [10*, c11s2.4]

The quality of software products can be improved through preventative processes or 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 stakeholder 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-Act (PDCA), evolutionary delivery, kaizen, and quality function deployment (QFD) offer techniques to specify quality objectives and determine whether they are met. The Software Engineering Institute's IDEAL is another method [7*]. Quality management is now recognized by the *SWEBOK Guide* as an important discipline.

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 is solicited frequently by implementing proactive communication activities.

1.5. Software Safety

[9*, 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 systems is safety-critical. There are increasing numbers of applications of safety-critical software in a growing number of industries. 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 [11], and emerging processes, tools, and techniques for developing safetycritical software. The intent of these standards, tools, and techniques is to reduce the risk of injecting faults into the software and thus improve software reliability.

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 is included in software engineering environments and software test environments.

Three complementary 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 necessitate more thorough inspections of requirements, design, and code or the use of more formal analytical techniques. Another technique for managing and controlling software risk is building assurance cases. An assurance case is a reasoned, auditable artifact created to support the contention that its claim or claims are satisfied. It contains the following and their relationships: one or more claims about properties; arguments that logically link the evidence and any assumptions to the claims; and a body of evidence and assumptions supporting these arguments [12].

2. Software Quality Management Processes

Software quality management is the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction [13, 14]. SQM defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle.

SQM comprises four subcategories: software quality planning, software quality assurance (SQA), software quality control (SQC), and software process improvement (SPI). 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 defining the software quality processes to be used. SQA activities define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes [5]. SQC activities examine specific project artifacts (documents and executables) to determine whether they comply with standards established for the project (including requirements, constraints, designs, contracts, and plans). SQC evaluates intermediate products as well as the final products.

The fourth SQM category dealing with improvement has various names within the software industry, including SPI, software quality improvement, and software corrective and preventive action. The activities in this category seek to improve process effectiveness, efficiency, and other characteristics with the ultimate goal of improving software quality. Although SPI could be included in any of the first three categories, an increasing number of organizations organize SPI into a separate category that may span across many projects (see the Software Engineering Process KA).

Software quality processes consist of tasks and techniques to indicate how software plans (e.g., software management, development, quality management, or 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.

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 help improve product quality (see the Software Engineering Management KA for related material on risk management).

2.1. Software Quality Assurance

[7*, c4–c6, c11, c12, c26–27]

To quell a widespread misunderstanding, software quality assurance is not testing. software quality assurance (SQA) is a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate and produce software products of suitable quality for their intended purposes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project; that is, free from technical, managerial, and

financial pressures from the project [5]. SQA has two aspects: product assurance and process assurance, which are explained in section 2.3.

The software quality plan (in some industry sectors it is termed the software quality assurance plan) defines the activities and tasks employed to ensure that software developed for a specific product satisfies the project's established requirements and user needs within project cost and schedule constraints and is commensurate with project risks. The SQAP first ensures that quality targets are clearly defined and understood.

The SQA plan's quality activities and tasks are specified with their costs, resource requirements, objectives, and schedule in relation to related objectives in the software engineering management, software development, and software maintenance plans. The SQA plan should be consistent with the software configuration management plan (see the Software Configuration Management KA). The SQA plan identifies documents, standards, practices, and conventions governing the project and how these items are 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 for 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

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

As stated in [15],

> 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 to be met.

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 helps 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

[9*, c24s3] [16*]

Reviews and audit processes are broadly defined as static—meaning that no software programs or models are executed—examination of software engineering artifacts with respect to standards that have been established by the organization or project for those artifacts. Different types of reviews and audits are distinguished by their purpose, levels of independence, tools and techniques, roles, and by the subject of the activity. Product assurance and process assurance audits are typically conducted by software quality assurance (SQA) personnel who are independent of development

teams. Management reviews are conducted by organizational or project management. The engineering staff conducts technical reviews.

- Management reviews evaluate actual project results with respect to plans.
- Technical reviews (including inspections, walkthrough, and desk checking) examine engineering work-products.
- Process assurance audits. SQA process assurance activities make certain that the processes used to develop, install, operate, and maintain software conform to contracts, comply with any imposed laws, rules, and regulations and are adequate, efficient and effective for their intended purpose [5].
- Product assurance audits. SQA product assurance activities make certain to provide evidence that software products and related documentation are identified in and comply with contracts; and ensure that nonconformances are identified and addressed [5].

2.3.1. Management Reviews

As stated in [16*],

> The purpose of a management review is to monitor progress, determine the status of plans and schedules, and evaluate the effectiveness of management processes, tools and techniques. Management reviews compare actual project results against plans to determine the status of projects or maintenance efforts. The main parameters of management reviews are project cost, schedule, scope, and quality. Management reviews evaluate decisions about corrective actions, changes in the allocation of resources, or changes to the scope of the project.

Inputs to management reviews may include 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

As stated in [16*],

> 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.

Although any work-product can be reviewed, technical reviews are performed on the main software engineering work-products of software requirements and software design.

Purpose, roles, activities, and most importantly the level of formality distinguish different types of technical reviews. Inspections are the most formal, walkthroughs less, and pair reviews or desk checks are the least formal.

Examples of specific roles include a decision maker (i.e., software lead), a review leader, a recorder, and checkers (technical staff members who examine the work-products). Reviews are also distinguished by whether meetings (face to face or electronic) are included in the process. In some review methods checkers solitarily examine work-products and send their results back to a coordinator. In other methods checkers work cooperatively in meetings. A technical review may require 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 documented review procedure. The technical review is completed once all the activities listed in the examination have been completed.

Technical reviews of source code may include a wide variety of concerns such as analysis of algorithms, utilization of critical computer resources, adherence to coding standards, structure and

organization of code for testability, and safety-critical considerations.

Note that technical reviews of source code or design models such as UML are also termed static analysis (see topic 3, Practical Considerations).

2.3.3. Inspections

"The purpose of an inspection is to detect and identify software product anomalies" [16*]. Some important differentiators of inspections as compared to other types of technical reviews are these:

1. Rules. Inspections are based upon examining a work-product with respect to a defined set of criteria specified by the organization. Sets of rules can be defined for different types of workproducts (e.g., rules for requirements, architecture descriptions, source code).
2. Sampling. Rather that attempt to examine every word and figure in a document, the inspection process allows checkers to evaluate defined subsets (samples) of the documents under review.
3. Peer. Individuals holding management positions over members of the inspection team do not participate in the inspection. This is a key distinction between peer review and management review.
4. Led. An impartial moderator who is trained in inspection techniques leads inspection meetings.
5. Meeting. The inspection process includes meetings (face to face or electronic) conducted by a moderator according to a formal procedure in which inspection team members report the anomalies they have found and other issues.

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) checkers (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 (samples). Each team member examines the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique (see section 3.3.3) to a small section of the product or to the entire product with a focus on only one aspect—e.g., interfaces. During the inspection, the moderator conducts the session and verifies that everyone has prepared for the inspection and conducts the session. The inspection recorder documents anomalies found. A set of rules, with criteria 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 corresponds to one of the following options:

1. Accept with no or, at most, minor reworking
2. Accept with rework verification
3. Reinspect.

2.3.4. Walkthroughs

As stated in [16*],

> 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.

Walkthroughs are distinguished from inspections. The main difference is that the author presents the work-product to the other participants in a meeting (face to face or electronic). Unlike an inspection, the meeting participants may not have necessarily seen the material prior to the meeting. The meetings may be conducted less formally. The author takes the role of explaining and showing the material to participants and solicits feedback. Like inspections, walkthroughs may be conducted on any type of work-product including project plan, requirements, design, source code, and test reports.

2.3.5. Process Assurance and Product Assurance Audits

As stated in [16*],

> 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.

Process assurance audits determine the adequacy of plans, schedules, and requirements to achieve project objectives [5]. 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. Audits 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 [16*], 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

[9*, c11s1] [18*, c12] [17*, c15s3.2.2, c15s3.3.1, c16s9.10]

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 resides; the system functions could be safety-critical, mission-critical, business-critical, security-critical
- the physical environment in which the software system resides
- system and software functional (what the system does) and quality (how well the system performs its functions) 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 processes
- the intended users and use of the system
- the integrity level of the system.

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which of those resources 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, tools and techniques can be applied to reduce the risk of injecting faults into the intermediate deliverables or the final software product. 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

Defining integrity levels is a method of risk management.

> Software integrity levels are a range of values that represent software complexity, criticality, risk, safety level, security level,

> desired performance, reliability, or other project-unique characteristics that define the importance of the software to the user and acquirer. The characteristics used to determine software integrity level vary depending on the intended application and use of the system. The software is a part of the system, and its integrity level is to be determined as a part of that system.

The assigned software integrity levels may change as the software evolves. Design, coding, procedural, and technology features implemented in the system or software can raise or lower the assigned software integrity levels. The software integrity levels established for a project result from agreements among the acquirer, supplier, developer, and independent assurance authorities. A software integrity level scheme is a tool used in determining software integrity levels. [5]

As noted in [17*], "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]

Software quality evaluation (i.e., software quality control) techniques find defects, faults and failures. Characterizing these techniques leads to an understanding of the product, facilitates corrections to the process or the product, and informs management and other stakeholders of the status of the process or product. Many taxonomies exist and, while attempts have been made to gain consensus, the literature indicates that there are quite a few in use. Defect characterization is also used in audits and reviews, with the review leader often presenting a list of issues 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, may not be sufficient to identify the underlying causes of the defects. Specific types of problems need to be grouped to identify trends over time. The point is to establish a defect taxonomy that is meaningful to the organization and to software engineers.

Software quality control activities discover information at all stages of software development and maintenance. In some cases, the word *defect* is overloaded to refer to different types of anomalies. However, different engineering cultures and standards may use somewhat different meanings for these terms. The variety of terms prompts this section to provide a widely used set of definitions [19]:

- *Computational Error*: "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition."
- *Error*: "A human action that produces an incorrect result." A slip or mistake that a person makes. Also called human error.
- *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*: A defect in source code. An "incorrect step, process, or data definition in computer program." The encoding of a human error in source code. Fault is the formal name of a bug.
- *Failure*: An "event in which a system or system component does not perform a required function within specified limits." A failure is produced when a fault is encountered by the processor under specified conditions.

Using these definitions three widely used software quality measurements are defect density (number of defects per unit size of documents), fault density (number of faults per 1K lines of code), and failure intensity (failures per use-hour or per test-hour). Reliability models are built from failure data collected during software testing or from software in service and thus can be used to estimate the probability of 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 (e.g., find and fix bugs, create new build). Other activities attempt to eliminate

the causes of the defects—for example, root cause analysis (RCA). RCA activities include analyzing and summarizing the findings 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 software quality control techniques may be lost unless they are recorded. For some techniques (e.g., technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used (see topic 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

[7*, c7s3] [8*, c17] [9*, c12s5, c15s1, p417] [16*]

Software quality control 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.

3.3.1. Static Techniques

Static techniques examine software documentation (including requirements, interface specifications, designs, and models) and software source code without executing the code. There are many tools and techniques for statically examining software work-products (see section 2.3.2). In addition, tools that analyze source code control flow and search for dead code are considered to be static analysis tools because they do not involve executing the software code.

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 techniques involve executing the software code. 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. Code reading may utilize dynamic techniques. This discrepancy in categorizing indicates that people with different roles and experience in the organization may consider and apply these techniques differently.

Different groups may perform testing during software development, including groups independent of the development team. The Software Testing KA is devoted entirely to this subject.

3.3.3. Testing

Two types of testing may fall under 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 (third-party or IV&V) organization may be tasked to perform testing or to monitor the test process V&V may be called upon to evaluate the testing itself: adequacy of plans, processes, and procedures, and adequacy and accuracy of results.

The third party is not the developer, nor is it 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 (see the Software Testing KA).

3.4. Software Quality Measurement
[3*, c4] [8*, c17] [9*, p90]

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 Termination under section 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 (e.g., 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 (see the Engineering Foundations KA). These techniques include

- descriptive statistics based (e.g., Pareto analysis, run charts, scatter plots, normal distribution)
- statistical tests (e.g., the binomial test, chi-squared test)
- trend analysis (e.g., control charts; see *The Quality Toolbox* in the list of further readings)
- prediction (e.g., reliability models).

Descriptive statistics-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 and conduct process improvements where they appear to be most needed. Results from trend analysis may indicate that a schedule is being met, such as in testing, or that certain classes of faults may 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.

Software quality measurement includes measuring defect occurrences and applying statistical methods to understand the types of defects that occur most frequently. This information may be used by software process improvement for determining methods to prevent, reduce, or eliminate their recurrence. They also aid in understanding trends, how well detection and containment 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

Software quality tools include static and dynamic analysis tools. Static analysis tools input source code, perform syntactical and semantic analysis without executing the code, and present results to users. There is a large variety in the depth, thoroughness, and scope of static analysis tools that

can be applied to artifacts including models, in addition to source code. (See the Software Construction, Software Testing, and Software Maintenance KAs for descriptions of dynamic analysis tools.)

Categories of static analysis tools include the following:

- Tools that facilitate and partially automate reviews and inspections of documents and code. These tools can route work to different participants in order to partially automate and control a review process. They allow users to enter defects found during inspections and reviews for later removal.
- Some tools help organizations perform software safety hazard analysis. These tools provide, e.g., automated support for failure mode and effects analysis (FMEA) and fault tree analysis (FTA).
- Tools that support tracking of software problems provide for entry of anomalies discovered during software testing and subsequent analysis, disposition, and resolution. Some tools include support for 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 sometimes include the functionality to perform statistical analysis on data sets (for the purpose of discerning trends and making forecasts). Some of these tools provide defect and removal injection rates; defect densities; yields; distribution of defect injection and removal for each of the life cycle phases.