Software Quality Assurance

The Definition of Software Quality

Many attempts have been made to define quality, none of which are completely satisfying. However, one simple and quite widely accepted definition is that **quality** is the ability of a product or service to meet the evolving needs and desires of its stakeholders. Note first that this definition applies to both products and services, so it is quite broad. A **stakeholder** in something is anyone affected by or involved in or influencing it. The intuitive notion of quality is that it is about how "good" something is. Presumably, something is good or bad for someone, and who better to use as a standard for how good or bad something is than its stakeholders? But stakeholders are a broad group, so once more this definition casts a wide net in characterizing quality. Finally, note that the word *evolving* is present in this definition. The idea is that the quality of a product or service changes over time because the needs and desires of stakeholders change. The quality of something may decline (or increase) over time even if it has not changed because its stakeholders need or want something different from it. Hence quality must be evaluated continuously.

Software quality is the quality of a software product or service, that is, the ability of a software product or service to meet the evolving needs and desires of its stakeholders. Although pithy, this definition is somewhat vague. Consequently, many people have identified *dimensions* or *aspects* of software quality delineating particular ways in which a product or service succeeds or fails to meet stakeholder needs and desires. One specification of quality dimensions that has become particularly widespread was developed by the International Standards Organization (ISO) as part of its effort to define and promote quality assurance processes. The standard, designated ISO/IEC 25010 [1], defines software quality along eight dimensions: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

Functional suitability is the degree to which the product satisfies user needs. It includes the functional completeness, correctness, and appropriateness of the software product.

Performance efficiency is related to the processing times and throughput rates of the product, the resources used by the product, and the capacity of the product.

Compatibility is the degree to which the product can co-exist with and interoperate with other products.

Usability is the degree to which specified users can learn a product and actually use it to achieve specified goals to their satisfaction. It includes the ability of users to recognize that a product is appropriate for their needs, learn to use the system, and operate the system.

Reliability is the degree to which the product performs specified functions under specified conditions. It includes the availability of the product (the proportion of time that the product is operating properly when needed) and its ability to recover from interruptions.

Security is the degree to which the product provides confidentiality (access only to legitimate users), integrity (prevention of unauthorized modifications), authenticity (identifying users), non-repudiation (proving that changes have occurred), and accountability (ability to associate changes with particular agents).

Maintainability is the degree to which the product can be modified, improved, corrected, and adapted, and components of the product can be re-used.

Portability is the ease with which a product can be made to operate in a new or altered computing environment.

These dimensions of quality are standard and we will use this terminology throughout the book.

Quality Assurance

Quality assurance (QA) is a systematic pattern of activities intended to ensure that a product has adequate quality, that is, that it properly satisfies the needs and desires of its stakeholders. Though there is sometimes overlap, QA activities can be characterized as either validation-oriented or verification-oriented.[1]

> **Validation** is the process of determining if a product (or its specification) satisfies stakeholders' needs and desires.
>
> **Verification** is the process of determining if a product (or its specification) satisfies those needs and desires properly.

Validation is sometimes described as the process of answering the question "Are we building the proper or right product?" and verification is sometimes described as the process of answering the question "Are we building the product properly or right?"[2] In both cases, the objective is to determine if the product contains **defects**, which are any undesirable aspect of a product.

Perhaps the easiest way to understand the distinction between validation and verification is with an example. Suppose you are a newspaper reporter and you are told to write an article about homelessness (the product). You must both write about homelessness (the topic that you were given; the proper product) and you must write properly (use proper grammar, follow the newspaper's style guide, etc…). If you write an article about homelessness then your product can be validated. If you write an article properly then your product can be verified.

All validation activities are product-specific. In other words, validation is the process of determining if a specific product satisfies needs and desires. In the context of the newspaper example, an editor will not be able to validate your article if she/he does not know the product you were told to create.

On the other hand, some verification activities are product-specific and some are not. That is, some verification activities require information about the product and some do not. Again in the context of the newspaper example, a copy editor will be able to determine if your article is grammatical and follows the style guide without knowing what product you were told to create. However, an assignment editor will not be able to determine if your argument is compelling without knowing what product you were told to create.

Validation is sometimes described as involving customer satisfaction and verification is sometimes described as involving conformance to specifications. However, while intuitive, this is too simplistic. First, it ignores the fact that verification can occur at all phases of the process. That is, needs lists,

---

[1] Unfortunately, there are many different definitions of these terms. IEEE-610 defines validation as the "process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements" and verification is the "process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase". ANSI/EIA-632 defines validation as "confirmation by examination and provision of objective evidence that the specific intended use of a product … is accomplished in an intended usage environment and verification as "confirmation by examination and provision of objective evidence that the specified requirements to which a product was built, coded or assembled have been fulfilled".

[2] The word "proper" is used here, rather than the word "correct", to avoid confusion later. When referring to needs lists, the word "correct" is used to refer to items that accurately reflect a stakeholder's need and are prioritized appropriately. When referring to software requirements specifications, the word "correct" is used to refer to a product that is a reasonable compromise among conflicting needs and desires.

requirements specifications, engineering designs and code can (and should) all be verified. Second, it ignores the fact that some aspects of verification are independent of the particular product (e.g., readable code).

Note that it may be impossible to validate a product that can't be verified. Continuing with the example of the reporter, it may not be possible to determine whether the story is about the right topic if it is so ungrammatical as to be unreadable.

Note, also, that a product can fail to satisfy needs and desires properly for different reasons, and that these reasons may be readily apparent in some circumstances but difficult to ascertain in others. For example, consider a word processor in which the users want to have misspelled words rendered in red. Now suppose that the product, in fact, renders misspelled words in blue. Clearly, this is a simple matter of the wrong color being used. However, now suppose that the misspelled words are rendered in black. This could either be because spelling isn't being checked at all, the spell-checking algorithm is not working properly, or because the wrong color is being used.

Defect Elimination

Given that QA is the systematic pattern of activities intended to ensure that a product properly satisfies the needs and desires of its stakeholders, and that a defect is any undesirable aspect of a product, it is clear than an important aspect of QA is the elimination of defects. In general, defects can be eliminated in one of two ways: they can be detected and removed, or they can be prevented.

Defect detection and removal is easy to understand and, hence, widely used. Unfortunately, defect detection and removal has many drawbacks. First, while it is easy to understand, it is difficult and costly to do. Both detection and removal require special training, special tools, and time. Second, when too many defects are detected products may be cancelled. When defects go undetected customer satisfaction problems often arise. Hence relying only on defect detection and removal can lead to serious problems.

In contrast, when defects are prevented, then no cost and time are spent finding and fixing them, and there are no bad consequences from failing to find them (because they are not there to be found). Hence, defect prevention is almost always superior to defect detection and removal.

The Centrality of Processes

Defects can be prevented only if the processes delivering a product or service make it unlikely that defects will occur. Also, defects can be found and eliminated only if there are defect detection and removal tasks built into product or service delivery processes. Therefore, processes are central to achieving quality. **Process improvement** is the analysis and modification of a process to enhance its efficiency, productivity, or the quality of its output. Process improvement is the fundamental means for improving and sustaining quality.

The centrality of processes for quality means that quality assurance is process driven, working towards the continuous improvement of an organization's processes. But processes cannot be improved using a single method or technique, and improvement cannot be done by QA specialists alone. Instead, an entire organization must be committed to quality, using a variety of tools and techniques and involving everyone in the organization. Furthermore, because of the relative costs of prevention and defect detection and removal, quality improvement should be oriented towards defect prevention, while recognizing that some defects will occur and incorporating detection and removal steps as necessary.

Companies committed to quality assurance are data driven, but not narrowly. They collect data about products, but also about processes and customers. Ultimately, such companies are *customer focused*—no data is more important that data about customer satisfaction.

A successful quality culture involves periodic training and education for all members of the organization. It also involves specific evolving quality goals and the clear communication of those goals.

Defect Prevention

A variety of different tools and techniques have been developed that attempt to prevent defects in software products.

Of particular interest are the tools and techniques that can be used during multiple phases of the process. These tools and techniques can be thought of as **process guides**: standards and guidelines (e.g., documentation standards, programming style guides, natural language usage guides) that describe how people should behave as well as templates and checklists that make it easier to do so. Process guides are important because they prevent common mistakes from being made. However, they are also important because they lead to behaviors that reduce the number of uncommon mistakes.

Closely related, but somewhat more specific, are **analysis and design methodologies**. These are well-codified approaches to understanding and solving software-related problems. Perhaps the best known example is object-oriented analysis and design.

Another way to prevent defects is to use well-studied solutions to problems or, to use a colloquial expression, not reinventing the wheel. In software quality assurance this approach to defect prevention can take many forms. At a high level of abstraction, it includes the use of **reference architectures**, which are configurable architectures for an application domain. At a lower level of abstraction, it includes **design patterns**, which are customizable solutions to a design problems. At a still lower level of abstraction are well-studied **algorithms and data structures**. The most extreme example of this is the reuse of code and other assets.

To the extent that you must reinvent the wheel, there are two different approaches that can be taken to prevent defects in software. First, formal methods can be used to prove that a software product will behave correctly. These methods rely on the use of mathematical models to determine whether a program meets requirements either without executing the program or by simulating the execution of the program. Second, **prototypes** can be used to identify likely defects before they arise.

Throwaway/exploratory prototypes are developed to help the design process and then discarded; relatively little time and effort are devoted to them. **Evolutionary prototypes**, on the other hand, are created as part of an iterative process and ultimately become part of the final product; they must be well-designed and flexible.

Since defects arise over time, **version control** and **configuration management tools** can also be an effective way to prevent defects. These tools can be used to control change and ensure consistency in many different kinds of documents (e.g., requirements specifications, design documents, and code).

Similarly, **computer-aided software engineering** (CASE) tools can play an important role in preventing defects. These tools range from the fairly simple (e.g., color-coding programming language constructs) to the fairly sophisticated (e.g., round-trip integration of UML diagrams and implementations in code).

Finally, all defect prevention techniques discussed above rely on people knowing about them and using them correctly and effectively. Thus in some sense the most fundamental defect prevention technique of all is **training and education** in software processes, good practices, methods, and tools.

Defect Detection and Removal

Obviously, not all defects can be prevented. Hence, some aspects of SQA are concerned with the detection and removal of defects. These techniques can be classified as "review and correct" or "test and debug."

Review and Correct

Review and correct techniques can be used during all phases of the software engineering process. In many ways, review and correct techniques are very similar to one another; they vary principally in their levels of formality and in their degree of automation.

Automated tools include style and standards checkers (e.g., spelling and grammar checkers, and program syntax and style checkers) and consistency and completeness checkers. These will be discussed more fully in the chapter on implementation.

Manual techniques involve one or more individuals examining the work product (be it a requirements specification, design, or code). The most informal such technique is the **desk check**, which involves a single individual (usually the author of the product) looking for and correcting defects. Slightly more formal is the **walkthrough**, which involves one individual (usually the author of the product) guiding one or more others through the examination. For example, a walkthrough might be done by showing a group of developers some code or part of a design or requirements specification on an overhead projector and then having a discussion about any defects and how to correct them.

An **inspection** is a formal examination of a product by a team of trained inspectors. Because of their effectiveness and applicability to most software development work products, inspections are particularly important.

Each participant in an inspection has particular role. The process is overseen by a **moderator** who schedules and runs meetings, organizes and distributes the materials to be used, reports on results, and monitors follow-up activities. The **author** of the inspected work product is a meeting participant. The team meeting is guided by a **reader**. In some cases, the reader literally reads the product line-by-line; in other cases, the reader simply instructs the other inspectors to consider a particular part of the product. The **recorder** takes notes during inspection meetings. One inspector may be the recorder for all meetings or the recorder may change for each meeting. In general, the recorder should not be the moderator. The remaining participants are **inspectors** who check the work product beforehand and help find defects during the meeting.

The inspection process proceeds as follows.

1. **Readiness check**. The moderator ensures that the product has been through all other reviews (whether automated or manual) and any defects that were found have been corrected. In other words, at the time of an inspection, the product is thought to be free of defects.
2. **Overview meeting**. The author introduces and distributes the work product (and supporting materials) to the rest of the team. In many cases, this is done electronically and the team members do not actually meet.

3. **Preparation**. Each inspector (guided by a checklist) reviews the work product individually, making notes on either the checklist or the work product, as appropriate.
4. **Team inspection**. The reader guides the other inspectors through the work product. They comment on it (using the notes they made while preparing), and the recorder takes notes (recording the location of the problem, its nature, and its severity).
5. **Correction**. The moderator provides the results of the inspection to the author, who corrects all the defects.
6. **Follow-up**. The moderator ensures that the defects are, in fact, corrected, and determines if another inspection is necessary.

Inspections are most successful when the guidelines below are followed.

- Inspectors are guided by a good checklist. Checklists become good by being improved over time: items that don't help find defects are removed, and defects that are missed in inspections have items added to help find them.
- Information about the defects is specific.
- Inspectors attend at most one team meeting per day.
- Inspection meetings last at most two hours.
- The moderator is not a manager.
- The report and meeting interactions are not judgmental—attention is focussed on defects, not on the abilities of the author of the work product.
- The report is written and delivered to the author within 24 hours.

In 1976, Michael Fagan, who invented this technique and used it to inspect code, found that 67% of all faults found in a development effort were found in code inspections, and that an inspected product had 38% fewer faults after seven months in the field than a comparable product developed with walkthroughs. Later experiments and studies have confirmed the efficacy of inspections right down to today.

Test and Debug

While review and correct techniques can be used throughout all phases of the software engineering process, given the complexity of most software products, they are unlikely to detect and remove all defects. Hence, it is also necessary to test software.

> **Testing** is a validation and verification process that makes use of the system or product (including prototypes) while it is in operation or being operated on.

In the context of software products, the phrase "in operation" refers to an executable product (e.g., a word processor) whereas the phrase "being operated on" refers to an input to an executable product (e.g., a document template or stylesheet). Because software testing involves the product while it is "operating" or "being operated on," it is only useful in detecting certain kinds of defects. These defects are referred to as faults and can be defined in terms of failures as follows.

> A **failure** is a deviation between a product's actual behavior and its intended behavior (e.g., a feature that is missing or incorrect).
>
> A **fault** is a defect that could (or does) give rise to a failure.

In the testing of software products it is important to distinguish between cause and effect, and the manifestations of causes and effects. To that end, the following definition is often used.

A **trigger** is a condition that causes a fault to result in a failure.

Taking this into account, software testing can be defined more concretely as follows.

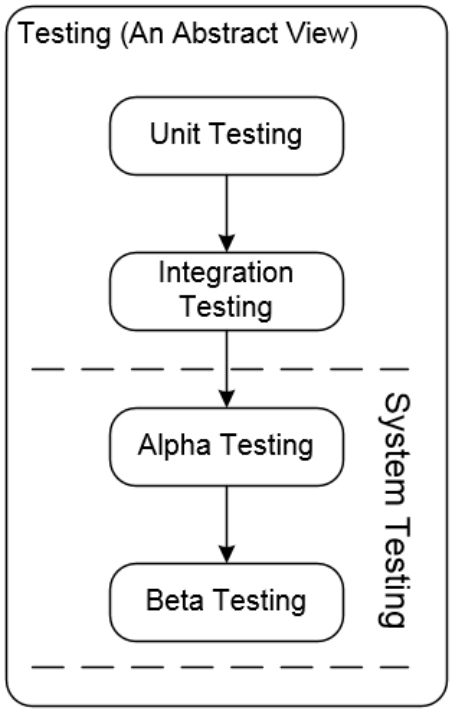
Software testing is the process of identifying failures by creating "inputs" and applying them to the software product.

A particular choice of "inputs" is referred to as a **test case** (or **test point**, or **test**). A collection of test cases or points is referred to as a **test suite**.

After testing has identified a fault it must be corrected. This process, which can be fairly complicated, is known as debugging.

Debugging is the process of using trigger conditions to identify and correct faults.

It is relatively inexpensive (compared to other engineering disciplines) to test software (after it has been implemented), but relatively expensive (compared to testing) to debug software. Hence, because it is easier to debug small units, software should be tested extensively and at various times throughout the process. The different kinds of tests and testing (the details of which are described in subsequent chapters) are often characterized as illustrated in the following diagram.



Unit testing exercises a single sub-program in isolation from any other sub-programs, **integration testing** exercises multiple sub-programs in combination, and **alpha** and **beta testing** (sometimes referred to collectively as **system testing**) exercise the entire software product.

Unit testing is conducted by the author of the sub-program and sometimes also dedicated testers (i.e., members of the organization whose main job is testing). Integration testing can involve the authors, testers of both. Alpha testing (which is sometimes subdivided into function testing and

performance testing) typically involves dedicated testers, and beta testing (which is sometimes subdivided into acceptance testing and installation testing) always involves users.

Obviously, a test can't be applied until after the relevant modules have been implemented. However, a test can be created before the relevant modules have been implemented. In fact, a test can be created before the relevant module has even been designed (in a detailed sense). As discussed below, the implications of this observation are somewhat different for traditional and agile processes.

Test and Debug in a Traditional Process

As noted in the chapter on requirements, in a traditional process, specifications should be testable or verifiable. Thus, the requirement specification is a good place to start when developing tests for a traditional process—each requirement should be mapped to one or more tests. Some requirements will be mapped to unit tests and others will be mapped to system tests. (In practice, few requirements tend to be mapped to integration tests.) While the requirements specification is a good place to start when developing tests, care must still be taken to ensure that the test suite is complete.

Unit testing is part of the implementation or construction phase. Hence, it tends to occur early and often in a traditional process. Unit authors are intimately involved with unit testing in a traditional process but, because of the length of cycles, testers are also often involved in the creation and execution of unit tests. Since integration testing and system testing occur much later, rigorous unit testing (and the debugging that results) is critical in a traditional process.

Integration testing often doesn't begin until fairly late in traditional processes. In other words, it is common for a large number of units to be constructed before they are combined and tested. As a result, in a traditional process it is often necessary to create **stubs**—placeholder code that stands in for units that have not yet been constructed. A stub has the same interface as the code it is standing in for, but provides none (or very little) of the required functionality. Unfortunately, when the stub is replaced with the fully-functional unit during integration testing, faults are often uncovered.

System testing often can't begin until the end of a traditional process. This is because traditional processes make no effort to have a working (if limited) version of the product at all times. Since the system can't be tested in its entirety until it exists, and it doesn't exist until the end of a traditional process, system testing can't occur until very late. Furthermore, traditional processes tend to have long cycle times, so there is typically a long period between system specification and system test. This is, perhaps, the biggest reason that traditional processes are not considered agile.

Because of the length of the cycles in traditional processes, the product released at the end of each cycle is often thought of as the "final product" (even though the product has a version number), and changes are described in relation to it. Of course, though one would like to think that all such changes are improvements, any change to a software product might introduce new faults. **Regression testing** is used to determine if changes to a product have introduced faults. In general, regression testing does not involve new tests. Rather, in regression testing the new product is subjected to the same tests as the original product.

Test and Debug in Scrum

In a traditional process, tests are created from the requirements specification. Since the Scrum process does not make use of a formal requirements specifications, test development in Scrum proceeds differently from test development in a traditional process.

Perhaps most importantly, in Scrum integration testing and system testing are conducted each and every sprint. Indeed, proponents of most agile processes argue that frequent integration and system

testing is a big part of what it means to be agile. In Scrum, integration and system testing are used to demonstrate that features in the product backlog have been completed.

Unit testing in Scrum is typically left entirely to the authors. Those who argue for this approach believe that unit testing will find implementation failures and integration and system testing will find design failures. Alternatively, one can specify *acceptance criteria* (also known as *conditions of satisfaction*) as unit tests. Using this approach, it becomes very clear when a user-level PBI has been completed.

The Efficiency of Defect Detection and Removal Techniques

The **defect removal efficiency** of an activity is the number of defects removed from a work product by the activity divided by the number of defects present in the work product at the start of the activity. Different techniques have different defect removal efficiencies. The numbers in the table below show defect removal efficiencies for several activities in a study from 2013 [2].

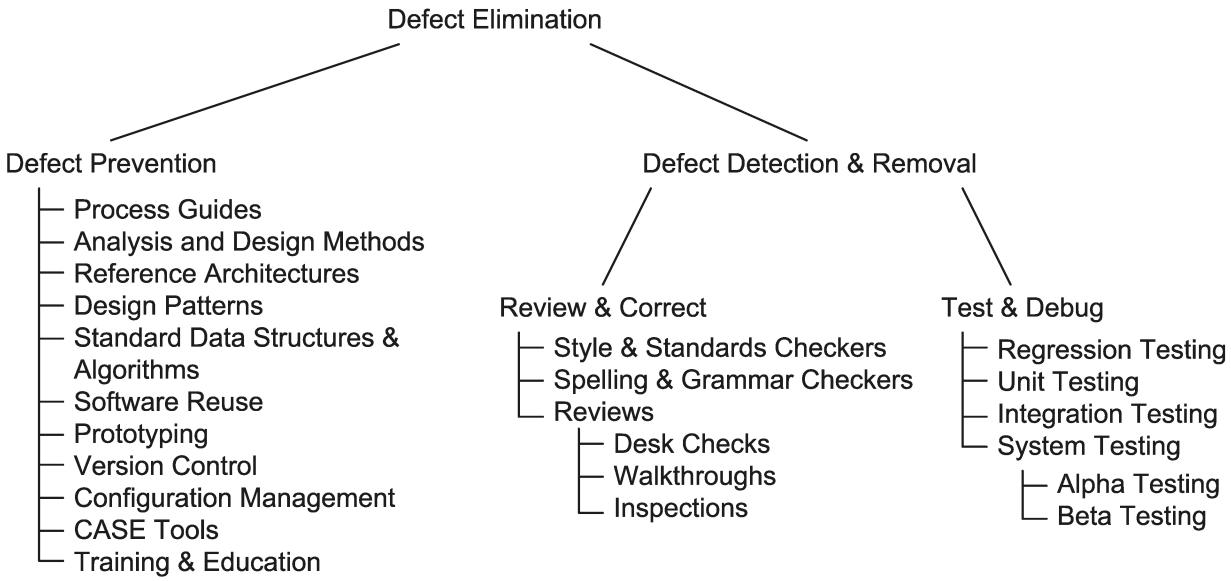
Technique	Minimum (%)	Median (%)	Maximum (%)
Requirements Review (informal)	20	30	50
Top-level design reviews (informal)	30	40	60
Detailed functional design inspection	30	65	80
Detailed logic design inspection	35	65	75
Code inspection or static analysis	35	60	90
Unit tests	10	25	50
Integration tests	25	45	60
System test	25	50	65
External Beta tests	15	40	75

It is important to note that different defect detection and removal techniques tend to find different kinds of defects, so a range of detection and removal techniques is needed. For example, requirements reviews tend to find failures to meet customer needs, while design reviews tend to find failures to meet requirements.

It is also important to note that, though many of these techniques have low efficiencies, because they are likely to be independent (in a probabilistic sense), using multiple techniques in sequence will likely produce a large defect removal efficiency. For example, if four independent defect detection and removal techniques each work with a probability of 0.6 (i.e., an efficiency of 60%) and they are used in sequence, the cumulative defect removal efficiency will be $1-(1-0.6)^4 \approx 97.5\%$.

A Defect Elimination Taxonomy

We have now discussed many defect elimination techniques, in various categories. The diagram below summarizes this material in a taxonomic tree.



References

1. ISO/IEC, *Software Product Quality*, http://iso25000.com/index.php/en/iso-25000-standards/iso-25010.
2. Jones, Capers. *Software Quality In 2013: A Survey Of The State Of The Art*. Http://Namcookanalytics.Com/Software-Quality-Survey-State-Art/.