Chapter 1

Software Quality

1.1 What is Quality?

The purpose of software quality analysis, or software quality engineering, is to produce acceptable products at acceptable cost, where cost includes calendar time (time to market, responsiveness to change requests) as well as conventional measures such as person-months and payroll cost. But what do we mean by quality? There is no simple, completely satisfactory answer, because like cost, quality has many different facets. We can begin to clarify the concept of quality analytically, by considering some of those facets.

Product and process qualities. Some qualities are properties of software products, while others are properties of the processes by which those products are created, supported, and evolved. For example, reliability and usability are product qualities, while visibility and time-to-market are process qualities. It would be folly to attempt to address the whole field of software process improvement as part of software quality engineering, and we will not do so here. Instead, we are concerned primarily with product qualities as the goals of software quality engineering, and process qualities as means to achieve those goals. For example, development processes with a high degree of visibility are necessary for creation of products with a high degree of reliability. The process goals with which software quality engineering is directly concerned are often on the "cost" side of the ledger. For example, we might have to weigh stringent reliability objectives against their impact on time-to-market, or seek ways to improve time-to-market without adversely impacting robustness.

A plethora of terms have been used to describe process aspects of software quality engineering, including software quality analysis, software quality control, software quality assurance, and software quality engineering. Among these, the only distinction we find really useful is between software quality control and all the other terms. Quality control is specifically a process of measuring differences between results and goals and providing feedback to the production process; quality analysis, or assurance, or engineering includes but is not limited to quality control.

Software product qualities: Internal and external. Software product qualities can be divided into those that are directly visible to a customer or client, and those that primarily affect the software development organization. Reliability, for example, is directly visible to the client. Maintainability primarily affects the development organization, although its consequences may indirectly affect the client as well, for example by increasing the time between product releases. Properties like dependability, latency, usability, and throughput, which are directly visible to users of a software product, are called *external* properties. Properties like maintainability, reusability, and traceability which are not directly visible in the product are called *internal* properties, even when their impact on the software development and evolution processes may indirectly affect users.

The external properties of software can ultimately be divided into dependability (does the software do what it is intended to do?) and usefulness. There is no precise way to distinguish these, but a rule of thumb is that when software is not dependable, we say it has a fault, or a defect, or (most often) a bug, resulting in an undesirable behavior or failure.

1.1.1 Dependability

Correctness. The simplest of the dependability properties is correctness: A program or system is correct if it is consistent with its specification. By definition, a specification divides all possible system behaviors[1] into two classes, *successes* (or correct executions) and *failures*. All of the possible behaviors of a correct system are successes.

Correctness is an all-or-nothing proposition. A program cannot be mostly correct or somewhat correct or 30% correct, it is absolutely correct on all possible behaviors or else it is not correct. It is very easy to achieve correctness, since every program is correct with respect to some (very bad) specification. Achieving correctness with respect to a useful specification, on the other hand, is seldom practical for non-trivial systems. Therefore, while correctness may be a noble goal, we are often interested in assessing some more achievable level of dependability.

Reliability. Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time. Like correctness, reliability is relative to a specification (which determines whether a unit of behavior is counted as a success or failure). Unlike correctness, reliability is also relative to a particular usage profile. The same program can be more or less reliable depending on how it is used.

Q: We have stated that 100% reliability is indistinguishable from correctness, but they are not quite identical. Under what circumstance might an incorrect program be 100% reliable? Hint: *Recall that a program may be more or less reli-*

[1] We are simplifying matters somewhat by considering only specifications of behaviors. A specification may also deal with other properties, such as the disk space required to install the application, and a system may thus be "incorrect" also if it violates one of these static properties.

able depending on how it is used, but a program is either correct or incorrect regardless of usage.

Particular measures of reliability can be used for different units of execution and different ways of counting success and failure. *Availability* is an appropriate measure when a failure has some duration in time. For example, a failure of a network router may make it impossible to use some functions of a local area network until the service is restored; between initial failure and restoration we say the router is "down" or "unavailable." The availability of the router is the proportion of time in which the system is "up" (providing normal service) as a fraction of total time. Thus, a network router that averages 1 hour of failure in each 24 hour period would have an availability of $\frac{23}{24}$, or 95.8%.

Q: We might measure the reliability of a network router as the fraction of all packets that are correctly routed, or as the fraction of total service time in which packets are correctly routed. When might these two measures be different? When might availability be the more useful measure of dependability? When might availability be less useful than the other measure of reliability?

Mean time between failures (MTBF) is yet another measure of reliability, also using time as the unit of execution. The hypothetical network switch that typically fails once in a 24 hour period and takes about an hour to recover has a mean time between failures of 23 hours. Note that availability does not distinguish between two failures of 30 minutes each and one failure lasting an hour, while MTBF does.

Q: If I am downloading a very large file over a slow modem, do I care more about the availability of my internet service provider or its mean time between failures?

Safety. The definitions of correctness and reliability have (at least) two major weaknesses. First, since the success or failure of an execution is relative to a specification, they are only as strong as the specification. Second, they make no distinction between a failure which is a minor annoyance and a failure which results in catastrophe. These are simplifying assumptions that we accept for the sake of precision, but in some circumstances — particularly, but not only, for critical systems — it is important to consider dependability properties that are less dependent on specification and which do distinguish among failures depending on severity.

Software safety is an extension of the well-established field of system safety into software. Safety is concerned with preventing certain undesirable behaviors, called *hazards.* It is quite explicitly not concerned with achieving any useful behavior apart from whatever functionality is needed to prevent hazards. Software safety is typically a concern in "critical" systems such as avionics and medical systems, but the basic principles apply to any system in which particularly undesirable behaviors can be distinguished from run-of-the-mill failure. For example, while it is annoying when my word processor crashes, it is much more annoying if it irrecoverably corrupts my document files, so the developers of a word processor might consider safety with respect

to the hazard of file corruption separately from reliability with respect to the complete functional requirements for the word processor.

Just as correctness is meaningless without a specification of allowed behaviors, safety is meaningless without a specification of hazards to be prevented, and in practice the first step of safety analysis is always finding and classifying hazards. Typically hazards are associated with some system in which the software is embedded (e.g., the medical device), rather than the software alone. The distinguishing feature of safety is that it is concerned *only* with these hazards, and not with other aspects of correct functioning.

The concept of safety is perhaps easier to grasp with familiar physical systems. For example, lawn-mowers in the U.S. are equipped with an interlock device, sometimes called a "dead-man switch." If this switch is not actively held by the operator, the engine shuts off. The dead-man switch does not contribute in any way to cutting grass; it's sole purpose is to prevent the operator from reaching into the mower blades while the engine runs.

One is tempted to say that safety is an aspect of correctness, because a good system specification would rule out hazards. However, safety is best considered as a quality quite distinct from correctness and reliability for two reasons. First, by focusing on a few hazards and ignoring other functionality, a separate safety specification can be much simpler than a complete system specification, and therefore easier to verify. To put it another way, while a good system specification *should* rule out hazards, we cannot be confident that either or specifications or our attempts to verify systems are good enough to provide the degree of assurance we require for hazard avoidance. Second, even if the safety specification were redundant with regard to the full system specification, it is important because (by definition) we regard avoidance of hazards as more crucial than satisfying other parts of the system specification.

Q: Can a system be correct and yet unsafe?

Q: Under what circumstances can making a system more safe make it less reliable?

1.1.2 Usefulness

It is quite possible to build systems that are very reliable, relatively free from hazards, and completely unusable. They may be unbearably slow, or have terrible user interfaces and unfathomable documentation, or be missing several crucial features. How should these properties be considered in software quality?

One answer is that they are not part of quality at all unless they have been explicitly specified, since quality is the presence of specified properties. However, a company whose products are rejected by its customers will take little comfort in knowing that, by some definitions, they were high quality products.

We can do better by considering quality as fulfillment of required and desired properties, as distinguished from specified properties. For example, even if a client does not explicitly specify the required performance of a system, there is always *some* level of performance which is required to be useful.

One of the most critical tasks in software quality analysis is making desired properties explicit, since properties that remain unspecified (even informally) are very likely

to surface unpleasantly when it is discovered that they are not met. In many cases these implicit requirements can not only be made explicit, but also made sufficiently precise that they can be made part of dependability or reliability. For example, while it is better to explicitly recognize usability as a requirement than to leave it implicit, it is better yet to augment[2] usability requirements with specific interface standards, so that a deviation from the standards is recognized as a defect.

1.2 Software Quality Analysis

We have already stated that the purpose of software quality analysis is to produce acceptable products at acceptable cost, and we have attempted some analysis of the properties that might bear on whether a product is acceptable. However, almost every software development activity either has some direct affect on these qualities or at least affects our ability to achieve them cost-effectively. What then should we include under the rubric of software quality analysis, and what exclude? Since specifications are crucial to software quality, should software specification be a sub-topic of quality analysis? Since process and product quality are intertwined, should software process be a sub-topic of software quality? We have already answered with respect to process: We will address some aspects of software process, but only as a means to other ends. We will take the same approach to requirements specification and many other software development activities, addressing parts of them as they pertain to software quality but leaving greater parts unaddressed.

It is often said that quality cannot be tested into a product at the end of development; it must be built in from the beginning. Nonetheless, validation and verification are a large part of the focus of software quality analysis; much of the "building in" is in laying the groundwork for validation and verification (V&V), e.g., ensuring that requirements have been specified and the system has been designed in a manner that makes V&V possible and cost-effective. Nor does a focus on V&V imply concentrating software quality analysis in the final stages of a software development process. On the contrary, not only are many V&V activities carried out in early stages (and on throughout product evolution as well), but the *process* part of software quality analysis is particularly concerned with choosing appropriate and effective V&V activities at each stage.

1.2.1 Validation and Verification

Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is called "validation." Fulfilling requirements is not the same as conforming to a requirements specification. In the first place, a specification is a statement about a particular proposed solution to a problem, while

[2] Interface standards augment, rather than replace usability requirements, because conformance to the standards is not sufficient assurance that the requirement is met. This is the same relation that other specifications have to the user requirements they are intended to fulfill. In general, verifying conformance to specifications does not replace validating satisfaction of requirements.

a well-written requirements statement should be general enough to admit of different solutions. More fundamentally, specifications are written by people, and therefore contain mistakes.

"Verification" is checking the consistency of an implementation to a specification. Here, "specification" and "implementation" are roles, not particular artifacts. For example, an overall design could play the role of "specification" and a more detailed design could play the role of "implementation;" checking whether the detailed design was consistent with the overall design would then be verification of the detailed design. Later, the same detailed design could play the role of "specification" with respect to source code, which would be verified against the design. In every case, though, verification is a check of consistency between two formal descriptions, in contrast to validation which compares a formal description (whether a requirements specification, a design, or a running system) against actual needs.

Validation against actual requirements necessarily involves human judgment and the potential for ambiguity, misunderstanding, and disagreement. In contrast, a specification should be sufficiently precise and unambiguous that there can be no disagreement about whether a particular system behavior is correct. Including a statement of a property in a requirements specification document does not render it verifiable. For example, statements like "The system shall be controlled through a user-friendly graphical user interface" are inherently unverifiable, although they can be validated.

Actual user requirements can almost never be fully formalized. Instead, a specification contains statements of properties which are intended collectively to fulfill the requirements. Some examples:

1.3 Exercises

Q: Software application domains can be characterized by the relative importance of schedule (calendar time), total cost, and dependability. For example, while all three are important for game software, schedule (shipping product in September to be available for holiday purchases) has particular weight, while dependability can be somewhat relaxed. Characterize a domain you are familiar with in these terms.

Q: Consider safety analysis for a software application domain with which you are familiar. What (if any) are the "hazards" that are sufficiently important to be considered separately from the main specification of required functionality? How are they typically addressed?

Q: We might have avoided a good deal of trouble by defining software quality analysis existentially, i.e., software quality analysis is what software quality analysts do. What would that mean in your organization? Would that definition have left out activities that, in your view, ought to be included in the scope of software quality analysis or software quality engineering?