DAY 4: Software Quality Landscape

External Characteristics

Characteristics that a user of the software product is aware of

501 **Adaptability** The extent to which a system can be used, without modification, in applications or environments other than those for which it was specifically designed.

501 **Robustness** The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

501 **Accuracy** The degree to which a system, as built, is free from error, especially with respect to quantitative outputs.

Accuracy differs from correctness; it is a determination of how well a system does the job it's built for rather than whether it was built correctly.

500 **Correctness** The degree to which a system is free from faults in its specification, design, and implementation.

500 **Usability** The ease with which users can learn and use a system.

501 **Efficiency** Minimal use of system resources, including memory and execution time.

501 **Reliability** The ability of a system to perform its required functions under stated conditions whenever required; having a long mean time between failures

501 **Integrity** The degree to which a system prevents unauthorized or improper access to its programs and its data

Internal Characteristics

Programmers care about the internal characteristics of the software as well as the external ones

501 **Portability** The ease with which you can modify a system to operate in an environment different from that for which it was specifically designed.

501 **Maintainability** The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.

501 **Flexibility** The extent to which you can modify a system for uses or environments other than those for which it was specifically designed.

501 **Reusability** The extent to which and the ease with which you can use parts of a system in other systems.

501 **Readability** The ease with which you can read and understand the source code of a system, especially at the detailed-statement level.

502 **Testability** The degree to which you can unit-test and system-test a system; the degree to which you can verify that the system meets its requirements.

502 **Understandability** The ease with which you can comprehend a system at both the system-organizational and detailed-statement levels. Understandability has to do with the coherence of the system at a more general level than readability does.

Improving Software Quality

504 **External Audits**

An external audit is a specific kind of technical review used to deter- mine the status of a project or the quality of a product being developed. An audit team is brought in from outside the organization and reports its findings to whoever com- missioned the audit, usually management

503 **Software Quality Objectives**

Set explicit quality objectives from external and internal characteristics

504 **Testing Strategy**

developing a test strategy in conjunction with the product requirements, architecture, and design

503 **Quality Assurance Activity**

One common problem in assuring quality is that quality is perceived as a secondary goal

504 **Engineering Guidelines**

Guidelines should control the technical character of the software as it's developed. Such guidelines apply to all software development activities, including problem definition, requirements development, architecture, con- struction, and system testing.

504 **Informal Technical Reviews**

Review your work before turn- ing it over for formal review. Desk-check the design of the code by walking through the code with a few peers

504 **Formal Technical Reviews**

Catch problems at the lowest-value stage where problems cost the least to correct.

504 **Quality Gates** are usually used to transition between requirements develop- ment and architecture, architecture and construction, and construction and system testing

504 A Quality Gate can be an **inspection**, a **peer** or **customer review**, or an **audit**.

504 **Does not need to be 100 percent complete**. Use the gate to determine if requirements / architecture are **good enough** to support further development.

Diagrams

503

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Helps it ↑
Hurts it ↓

Figure 20-1 Focusing on one external characteristic of software quality can affect other characteristics positively, adversely, or not at all.

509

Table 20-3 Extreme Programming's Estimated Defect-Detection Rate

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews (pair programming)	25%	35%	40%
Informal code reviews (pair programming)	20%	25%	35%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
Expected cumulative defect-removal efficiency	**~74%**	**~90%**	**~97%**

507

Table 20-2 Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

Chapter 20

The Software-Quality Landscape

cc2e.com/2036

Contents

Related Topics

- Collaborative construction: Chapter 21
- Developer testing: Chapter 22
- Debugging: Chapter 23
- Prerequisites to construction: Chapters 3 and 4
- Do prerequisites apply to modern software projects?: in Section 3.1

This chapter surveys software-quality techniques from a construction point of view. The entire book is about improving software quality, of course, but this chapter focuses on quality and quality assurance per se. It focuses more on big-picture issues than it does on hands-on techniques. If you're looking for practical advice about collaborative development, testing, and debugging, move on to the next three chapters.

20.1 Characteristics of Software Quality

Software has both external and internal quality characteristics. External characteristics are characteristics that a user of the software product is aware of, including the following:

- **Correctness** The degree to which a system is free from faults in its specification, design, and implementation.
- **Usability** The ease with which users can learn and use a system.

- **Efficiency** Minimal use of system resources, including memory and execution time.
- **Reliability** The ability of a system to perform its required functions under stated conditions whenever required—having a long mean time between failures.
- **Integrity** The degree to which a system prevents unauthorized or improper access to its programs and its data. The idea of integrity includes restricting unauthorized user accesses as well as ensuring that data is accessed properly—that is, that tables with parallel data are modified in parallel, that date fields contain only valid dates, and so on.
- **Adaptability** The extent to which a system can be used, without modification, in applications or environments other than those for which it was specifically designed.
- **Accuracy** The degree to which a system, as built, is free from error, especially with respect to quantitative outputs. Accuracy differs from correctness; it is a determination of how well a system does the job it's built for rather than whether it was built correctly.
- **Robustness** The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

Some of these characteristics overlap, but all have different shades of meaning that are applicable more in some cases, less in others.

External characteristics of quality are the only kind of software characteristics that users care about. Users care about whether the software is easy to use, not about whether it's easy for you to modify. They care about whether the software works correctly, not about whether the code is readable or well structured.

Programmers care about the internal characteristics of the software as well as the external ones. This book is code-centered, so it focuses on the internal quality characteristics, including

- **Maintainability** The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.
- **Flexibility** The extent to which you can modify a system for uses or environments other than those for which it was specifically designed.
- **Portability** The ease with which you can modify a system to operate in an environment different from that for which it was specifically designed.
- **Reusability** The extent to which and the ease with which you can use parts of a system in other systems.
- **Readability** The ease with which you can read and understand the source code of a system, especially at the detailed-statement level.

- **Testability** The degree to which you can unit-test and system-test a system; the degree to which you can verify that the system meets its requirements.
- **Understandability** The ease with which you can comprehend a system at both the system-organizational and detailed-statement levels. Understandability has to do with the coherence of the system at a more general level than readability does.

As in the list of external quality characteristics, some of these internal characteristics overlap, but they too each have different shades of meaning that are valuable.

The internal aspects of system quality are the main subject of this book and aren't discussed further in this chapter.

The difference between internal and external characteristics isn't completely clear-cut because at some level internal characteristics affect external ones. Software that isn't internally understandable or maintainable impairs your ability to correct defects, which in turn affects the external characteristics of correctness and reliability. Software that isn't flexible can't be enhanced in response to user requests, which in turn affects the external characteristic of usability. The point is that some quality characteristics are emphasized to make life easier for the user and some are emphasized to make life easier for the programmer. Try to know which is which and when and how these characteristics interact.

The attempt to maximize certain characteristics inevitably conflicts with the attempt to maximize others. Finding an optimal solution from a set of competing objectives is one activity that makes software development a true engineering discipline. Figure 20-1 shows the way in which focusing on some external quality characteristics affects others. The same kinds of relationships can be found among the internal characteristics of software quality.

The most interesting aspect of this chart is that focusing on a specific characteristic doesn't always mean a tradeoff with another characteristic. Sometimes one hurts another, sometimes one helps another, and sometimes one neither hurts nor helps another. For example, correctness is the characteristic of functioning exactly to specification. Robustness is the ability to continue functioning even under unanticipated conditions. Focusing on correctness hurts robustness and vice versa. In contrast, focusing on adaptability helps robustness and vice versa.

The chart shows only typical relationships among the quality characteristics. On any given project, two characteristics might have a relationship that's different from their typical relationship. It's useful to think about your specific quality goals and whether each pair of goals is mutually beneficial or antagonistic.

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Helps it ↑

Hurts it ↓

Figure 20-1 Focusing on one external characteristic of software quality can affect other characteristics positively, adversely, or not at all.

20.2 Techniques for Improving Software Quality

Software quality assurance is a planned and systematic program of activities designed to ensure that a system has the desired characteristics. Although it might seem that the best way to develop a high-quality product would be to focus on the product itself, in software quality assurance you also need to focus on the software-development process. Some of the elements of a software-quality program are described in the following subsections:

Software-quality objectives One powerful technique for improving software quality is setting explicit quality objectives from among the external and internal characteristics described in the previous section. Without explicit goals, programmers might work to maximize characteristics different from the ones you expect them to maximize. The power of setting explicit goals is discussed in more detail later in this section.

Explicit quality-assurance activity One common problem in assuring quality is that quality is perceived as a secondary goal. Indeed, in some organizations, quick and dirty programming is the rule rather than the exception. Programmers like Global Gary, who litter their code with defects and "complete" their programs quickly, are rewarded more than programmers like High-Quality Henry, who write excellent programs and make sure that they are usable before releasing them. In such organizations, it shouldn't be surprising that programmers don't make quality their first priority. The organization must show programmers that quality is a priority. Making the quality-assurance activity explicit makes the priority clear, and programmers will respond accordingly.

Cross-Reference For details on testing, see Chapter 22, "Developer Testing."

Testing strategy Execution testing can provide a detailed assessment of a product's reliability. Part of quality assurance is developing a test strategy in conjunction with the product requirements, architecture, and design. Developers on many projects rely on testing as the primary method of both quality assessment and quality improvement. The rest of this chapter demonstrates in more detail that this is too heavy a burden for testing to bear by itself.

Cross-Reference For a discussion of one class of software-engineering guidelines appropriate for construction, see Section 4.2, "Programming Conventions."

Software-engineering guidelines Guidelines should control the technical character of the software as it's developed. Such guidelines apply to all software development activities, including problem definition, requirements development, architecture, construction, and system testing. The guidelines in this book are, in one sense, a set of software-engineering guidelines for construction.

Informal technical reviews Many software developers review their work before turning it over for formal review. Informal reviews include desk-checking the design or the code or walking through the code with a few peers.

Cross-Reference Reviews and inspections are discussed in Chapter 21, "Collaborative Construction."

Formal technical reviews One part of managing a software-engineering process is catching problems at the "lowest-value" stage—that is, at the time at which the least investment has been made and at which problems cost the least to correct. To achieve such a goal, developers use "quality gates," periodic tests or reviews that determine whether the quality of the product at one stage is sufficient to support moving on to the next. Quality gates are usually used to transition between requirements development and architecture, architecture and construction, and construction and system testing. The "gate" can be an inspection, a peer review, a customer review, or an audit.

Cross-Reference For more details on how development approaches vary depending on the kind of project, see Section 3.2, "Determine the Kind of Software You're Working On."

A "gate" does not mean that architecture or requirements need to be 100 percent complete or frozen; it does mean that you will use the gate to determine whether the requirements or architecture are good enough to support downstream development. "Good enough" might mean that you've sketched out the most critical 20 percent of the requirements or architecture, or it might mean you've specified 95 percent in excruciating detail—which end of the scale you should aim for depends on the nature of your specific project.

External audits An external audit is a specific kind of technical review used to determine the status of a project or the quality of a product being developed. An audit team is brought in from outside the organization and reports its findings to whoever commissioned the audit, usually management.

Development Process

Further Reading For a discussion of software development as a process, see *Professional Software Development* (McConnell 1994).

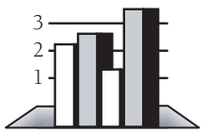
Each of the elements mentioned so far has something to do explicitly with assuring software quality and implicitly with the process of software development. Development efforts that include quality-assurance activities produce better software than those that do not. Other processes that aren't explicitly quality-assurance activities also affect software quality.

point in the project. Table 20-2 shows the percentages of defects detected by several common defect-detection techniques.

Table 20-2 Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).



HARD DATA

The most interesting facts that this data reveals is that the modal rates don't rise above 75 percent for any single technique and that the techniques average about 40 percent. Moreover, for the most common kinds of defect detection—unit testing and integration testing—the modal rates are only 30–35 percent. The typical organization uses a test-heavy defect-removal approach and achieves only about 85 percent defect-removal efficiency. Leading organizations use a wider variety of techniques and achieve defect-removal efficiencies of 95 percent or higher (Jones 2000).

The strong implication is that if project developers are striving for a higher defect-detection rate, they need to use a combination of techniques. A classic study by Glenford Myers confirmed this implication (1978b). Myers studied a group of programmers with a minimum of 7 and an average of 11 years of professional experience. Using a program with 15 known errors, he had each programmer look for errors by using one of these techniques:

- Execution testing against the specification
- Execution testing against the specification with the source code
- Walk-through/inspection using the specification and the source code

Table 20-3 Extreme Programming's Estimated Defect-Detection Rate

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews (pair programming)	25%	35%	40%
Informal code reviews (pair programming)	20%	25%	35%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
Expected cumulative defect-removal efficiency	**~74%**	**~90%**	**~97%**

Cost of Finding Defects

Some defect-detection practices cost more than others. The most economical practices result in the least cost per defect found, all other things being equal. The qualification that all other things must be equal is important because per-defect cost is influenced by the total number of defects found, the stage at which each defect is found, and other factors besides the economics of a specific defect-detection technique.

HARD DATA

Most studies have found that inspections are cheaper than testing. A study at the Software Engineering Laboratory found that code reading detected about 80 percent more faults per hour than testing (Basili and Selby 1987). Another organization found that it cost six times as much to detect design defects by using testing as by using inspections (Ackerman, Buchwald, and Lewski 1989). A later study at IBM found that only 3.5 staff hours were needed to find each error when using code inspections, whereas 15–25 hours were needed to find each error through testing (Kaplan 1995).

Cost of Fixing Defects

The cost of finding defects is only one part of the cost equation. The other is the cost of fixing defects. It might seem at first glance that how the defect is found wouldn't matter—it would always cost the same amount to fix.

Cross-Reference For details on the fact that defects become more expensive the longer they stay in a system, see "Appeal to Data" in Section 3.1. For an up-close look at errors themselves, see Section 22.4, "Typical Errors."

That isn't true because the longer a defect remains in the system, the more expensive it becomes to remove. A detection technique that finds the error earlier therefore results in a lower cost of fixing it. Even more important, some techniques, such as inspections, detect the symptoms and causes of defects in one step; others, such as testing, find symptoms but require additional work to diagnose and fix the root cause. The result is that one-step techniques are substantially cheaper overall than two-step ones.