CHAPTER

14 QUALITY CONCEPTS

KEY CONCEPTS

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised. Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, *CIO Magazine* [Lev01] trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do." *InformationWeek* [Ric01] echoed the same concern:

> Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn't include the cost of losing angry customers. Because IT shops write applications that rely on packaged infrastructure software, bad code can wreak havoc on custom apps as well. . . .
>
> Just how bad is bad software? Definitions vary, but experts say it takes only three or four defects per 1,000 lines of code to make a program perform poorly. Factor in that most programmers inject about one error for every 10 lines of code they write, multiply that by the millions of lines of code in many commercial products, then figure it costs software vendors at least half their development budgets to fix errors while testing. Get the picture?

QUICK LOOK

What is it? The answer isn't as easy as you might think. You know quality when you see it, and yet, it can be an elusive thing to define. But for computer software, quality is something that we must define, and that's what I'll do in this chapter.

Who does it? Everyone—software engineers, managers, all stakeholders—involved in the software process is responsible for quality.

Why is it important? You can do it right, or you can do it over again. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that it must do. That results in lower costs, and more importantly, improved time-to-market.

What are the steps? To achieve high-quality software, four activities must occur: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure.

What is the work product? Software that meets its customer's needs, performs accurately and reliably, and provides value to all who use it.

How do I ensure that I've done it right? Track quality by examining the results of all quality control activities, and measure quality by examining errors before delivery and defects released to the field.

In 2005, *ComputerWorld* [Hil05] lamented that "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction. A year later, *InfoWorld* [Fos06] wrote about the "the sorry state of software quality" reporting that the quality problem had not gotten any better.

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both*—and that's the problem. In this chapter, I consider software quality as a concept and examine why it's worthy of serious consideration whenever software engineering practices are applied.

14.1 What Is Quality?

In his mystical book, *Zen and the Art of Motorcycle Maintenance,* Robert Persig [Per74] commented on the thing we call *quality*:

> Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others; that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Indeed—what is it?

What are the different ways in which quality can be viewed?

At a somewhat more pragmatic level, David Garvin [Gar84] of the Harvard Business School suggests that "quality is a complex and multifaceted concept" that can be described from five different points of view. The *transcendental view* argues (like Persig) that quality is something that you immediately recognize, but cannot explicitly define. The *user view* sees quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality. The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality. The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product. Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and

greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications.

In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

Quote:

"People forget how fast you did a job—but they always remember how well you did it."

Howard Newton

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass [Gla98] argues that a more "intuitive" relationship is in order:

user satisfaction = compliant product + good quality + delivery within budget and schedule

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco [DeM98] reinforces this view when he states: "A product's quality is a function of how much it changes the world for the better." This view of quality contends that if a software product provides substantial benefit to its end users, they may be willing to tolerate occasional reliability or performance problems.

14.2 Software Quality

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *software* quality? In the most general sense, software quality can be defined[1] as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. An *effective software process* establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high-quality software. Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.
2. A *useful product* delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free

1 This definition has been adapted from [Bes04] and replaces a more manufacturing-oriented view presented in earlier editions of this book.

way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software.

3. By *adding value for both the producer and user* of a software product, high-quality software provides benefits for the software organization and the end-user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process. The end result is (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial for the business.

14.2.1 Garvin's Quality Dimensions

David Garvin [Gar87] suggests that quality should be considered by taking a multidimensional viewpoint that begins with an assessment of conformance and terminates with a transcendental (aesthetic) view. Although Garvin's eight dimensions of quality were not developed specifically for software, they can be applied when software quality is considered:

Performance quality. Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end user?

Feature quality. Does the software provide features that surprise and delight first-time end users?

Reliability. Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error-free?

Conformance. Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Durability. Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?

Serviceability. Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period? Can support staff acquire all information they need to make changes or correct defects? Douglas Adams [Ada93] makes a wry comment that seems appropriate here: "The difference

between something that can go wrong and something that can't possibly go wrong is that when something that can't possibly go wrong goes wrong it usually turns out to be impossible to get at or repair."

Aesthetics. There's no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.

Perception. In some situations, you have a set of prejudices that will influence your perception of quality. For example, if you are introduced to a software product that was built by a vendor who has produced poor quality in the past, your guard will be raised and your perception of the current software product quality might be influenced negatively. Similarly, if a vendor has an excellent reputation, you may perceive quality, even when it does not really exist.

Garvin's quality dimensions provide you with a "soft" look at software quality. Many (but not all) of these dimensions can only be considered subjectively. For this reason, you also need a set of "hard" quality factors that can be categorized in two broad groups: (1) factors that can be directly measured (e.g., defects uncovered during testing) and (2) factors that can be measured only indirectly (e.g., usability or maintainability). In each case measurement must occur. You should compare the software to some datum and arrive at an indication of quality.

14.2.2 McCall's Quality Factors

McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 14.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

Referring to the factors noted in Figure 14.1, McCall and his colleagues provide the following descriptions:

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

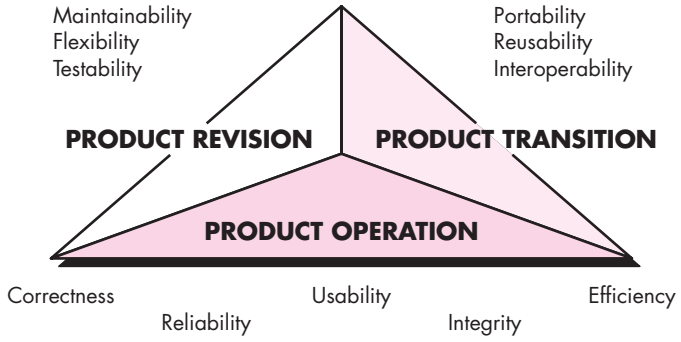
Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 25).]

Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. Extent to which access to software or data by unauthorized persons can be controlled.

Usability. Effort required to learn, operate, prepare input for, and interpret output of a program.

FIGURE 14.1
McCall's software quality factors



Quote:

"The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten."

Karl Weigers (unattributed quote)

Maintainability. Effort required to locate and fix an error in a program. [This is a very limited definition.]

Flexibility. Effort required to modify an operational program.

Testability. Effort required to test a program to ensure that it performs its intended function.

Portability. Effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

Interoperability. Effort required to couple one system to another.

It is difficult, and in some cases impossible, to develop direct measures[2] of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality.

14.2.3 ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.

Reliability. The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.

2 A *direct measure* implies that there is a single countable value that provides a direct indication of the attribute being examined. For example, the "size" of a program can be measured directly by counting the number of lines of code.

Usability. The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.

Maintainability. The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

ADVICE

Although it's tempting to develop quantitative measures for the quality factors noted here, you can also create a simple checklist of attributes that provide a solid indication that the factor is present.

Like other software quality factors discussed in the preceding subsections, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.

14.2.4 Targeted Quality Factors

Quote:

"Any activity becomes creative when the doer cares about doing it right, or better."

John Updike

The quality dimensions and factors presented in Sections 14.2.1 and 14.2.2 focus on the software as a whole and can be used as a generic indication of the quality of an application. A software team can develop a set of quality characteristics and associated questions that would probe[3] the degree to which each factor has been satisfied. For example, McCall identifies *usability* as an important quality factor. If you were asked to review a user interface and assess its usability, how would you proceed? You might start with the subattributes suggested by McCall—understandability, learnability, and operability—but what do these mean in a pragmatic sense?

To conduct your assessment, you'll need to address specific, measurable (or at least, recognizable) attributes of the interface. For example [Bro03]:

Intuitiveness. The degree to which the interface follows expected usage patterns so that even a novice can use it without significant training.

- Is the interface layout conducive to easy understanding?
- Are interface operations easy to locate and initiate?
- Does the interface use a recognizable metaphor?
- Is input specified to economize key strokes or mouse clicks?
- Does the interface follow the three golden rules? (Chapter 11)
- Do aesthetics aid in understanding and usage?

3 These characteristics and questions would be addressed as part of a software review (Chapter 15).

Efficiency. The degree to which operations and information can be located or initiated.

- Does the interface layout and style allow a user to locate operations and information efficiently?
- Can a sequence of operations (or data input) be performed with an economy of motion?
- Are output data or content presented so that it is understood immediately?
- Have hierarchical operations been organized in a way that minimizes the depth to which a user must navigate to get something done?

Robustness. The degree to which the software handles bad input data or inappropriate user interaction.

- Will the software recognize the error if data at or just outside prescribed boundaries is input? More importantly, will the software continue to operate without failure or degradation?
- Will the interface recognize common cognitive or manipulative mistakes and explicitly guide the user back on the right track?
- Does the interface provide useful diagnosis and guidance when an error condition (associated with software functionality) is uncovered?

Richness. The degree to which the interface provides a rich feature set.

- Can the interface be customized to the specific needs of a user?
- Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command?

As the interface design is developed, the software team would review the design prototype and ask the questions noted. If the answer to most of these questions is "yes," it is likely that the user interface exhibits high quality. A collection of questions similar to these would be developed for each quality factor to be assessed.

14.2.5 The Transition to a Quantitative View

In the preceding subsections, I have presented a variety of qualitative factors for the "measurement" of software quality. The software engineering community strives to develop precise measures for software quality and is sometimes frustrated by the subjective nature of the activity. Cavano and McCall [Cav78] discuss this situation:

> The determination of quality is a key factor in every day events—wine tasting contests, sporting events [e.g., gymnastics], talent contests, etc. In these situations, quality is judged in the most fundamental and direct manner: side by side comparison of objects under identical conditions and with predetermined concepts. The wine may be judged according to clarity, color, bouquet, taste, etc. However, this type of judgment is very subjective; to have any value at all, it must be made by an expert.

> Subjectivity and specialization also apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as a way to derive quantitative measurements of software quality for objective analysis. . . . Since there is no such thing as absolute knowledge, one should not expect to measure software quality exactly, for every measurement is partially imperfect. Jacob Bronkowski described this paradox of knowledge in this way: "Year by year we devise more precise instruments with which to observe nature with more fineness. And when we look at the observations we are discomfited to see that they are still fuzzy, and we feel that they are as uncertain as ever."

In Chapter 23, I'll present a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure *quality* but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

14.3 The Software Quality Dilemma

In an interview [Ven03] published on the Web, Bertrand Meyer discusses what I call the *quality dilemma:*

> If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway. Either you missed the market window, or you simply exhausted all your resources. So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete.

When you're faced with the quality dilemma (and everyone is faced with it at one time or another), try to achieve balance—enough effort to produce acceptable quality without burying the project.

It's fine to state that software engineers should strive to produce high-quality systems. It's even better to apply good practices in your attempt to do so. But the situation discussed by Meyer is real life and represents a dilemma for even the best software engineering organizations.

14.3.1 "Good Enough" Software

Stated bluntly, if we are to accept the argument made by Meyer, is it acceptable to produce "good enough" software? The answer to this question must be "yes," because major software companies do it every day. They create software with known bugs and deliver it to a broad population of end users. They recognize that some of the functions and features delivered in Version 1.0 may not be of the highest quality and plan for improvements in Version 2.0. They do this knowing that some customers will complain, but they recognize that time-to-market may trump better quality as long as the delivered product is "good enough."

Exactly what is "good enough"? Good enough software delivers high-quality functions and features that users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality.

This idea may resonate with many readers. If you're one of them, I can only ask you to consider some of the arguments against "good enough."

It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in. As I noted earlier, it can argue that it will improve quality in subsequent versions. By delivering a good enough version 1.0, it has cornered the market.

If you work for a small company be wary of this philosophy. When you deliver a good enough (buggy) product, you risk permanent damage to your company's reputation. You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.

If you work in certain application domains (e.g., real-time embedded software) or build application software that is integrated with hardware (e.g., automotive software, telecommunications software), delivering software with known bugs can be negligent and open your company to expensive litigation. In some cases, it can even be criminal. No one wants good enough aircraft avionics software!

So, proceed with caution if you believe that "good enough" is a short cut that can solve your software quality problems. It can work, but only for a few and only in a limited set of application domains.[4]

14.3.2 The Cost of Quality

The argument goes something like this—*we know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want.* On its face, this argument seems reasonable (see Meyer's comments earlier in this section). There is no question that quality has a cost, but lack of quality also has a cost—not only to end users who must live with buggy software, but also to the software organization that has built and must maintain it. The real question is this: *which cost should we be worried about?* To answer this question, you must understand both the cost of achieving quality and the cost of low-quality software.

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization must collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

4 A worthwhile discussion of the pros and cons of "good enough" software can be found in [Bre02].

ADVICE

Don't be afraid to incur significant prevention costs. Rest assured that your investment will provide an excellent return.

Prevention costs include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities.

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include:

- Cost of conducting technical reviews (Chapter 15) for software engineering work products
- Cost of data collection and metrics evaluation (Chapter 23)
- Cost of testing and debugging (Chapters 18 through 21)

Failure costs are those that would disappear if no errors appeared before or after shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when you detect an error in a product prior to shipment. Internal failure costs include

- Cost required to perform rework (repair) to correct an error
- Cost that occurs when rework inadvertently generates side effects that must be mitigated
- Costs associated with the collection of quality metrics that allow an organization to assess the modes of failure

Quote:

"It takes less time to do a thing right than to explain why you did it wrong."

H. W. Longfellow

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real. Bad things happen when low-quality software is produced.

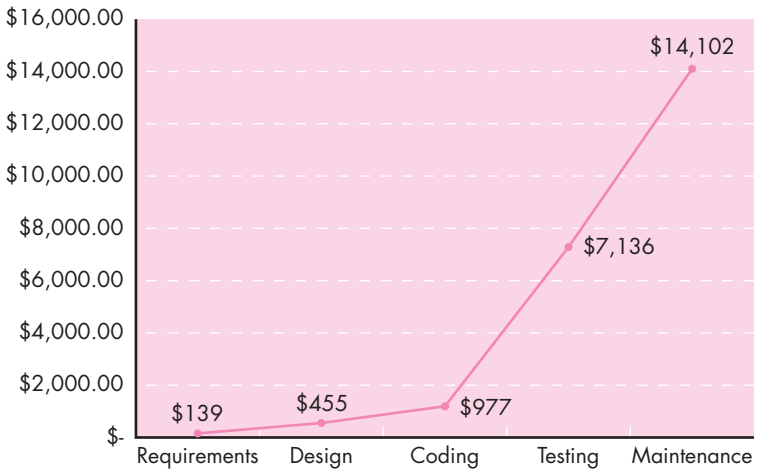
In an indictment of software developers who refuse to consider external failure costs, Cem Kaner [Kan95] states:

> Many of the external failure costs, such as goodwill, are difficult to quantify, and many companies therefore ignore them when calculating their cost-benefit tradeoffs. Other external failure costs can be reduced (e.g. by providing cheaper, lower-quality, post-sale support, or by charging customers for support) without increasing customer satisfaction. By ignoring the costs to our customers of bad products, quality engineers encourage quality-related decision-making that victimizes our customers, rather than delighting them.

As expected, the relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 14.2, based on data collected by Boehm and Basili [Boe01b] and illustrated by Cigital Inc. [Cig07], illustrates this phenomenon.

The industry average cost to correct a defect during code generation is approximately $977 per error. The industry average cost to correct the same error if it is

FIGURE 14.2
Relative cost of correcting errors and defects
Source: Adapted from [Boe01b].



discovered during system testing is $7,136 per error. Cigital Inc. [Cig07] considers a large application that has 200 errors introduced during coding.

> According to industry average data, the cost of finding and correcting defects during the coding phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 × $977) is approximately $195,400.
>
> Industry average data shows that the cost of finding and correcting defects during the system testing phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 × $7,136) would have been approximately $356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 × $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been $2,472,100 ($2,115,300 + $356,800).

Even if your software organization has costs that are half of the industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

14.3.3 Risks

In Chapter 1 of this book, I wrote "people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right." The implication is that low-quality software increases risks for both the developer and the end user. In the preceding subsection, I discussed one of these risks (cost). But the downside of poorly designed and implemented applications does not always stop with dollars and time. An extreme example [Gag04] might serve to illustrate.

Throughout the month of November 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, 5 of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute doses of radiation for each patient.

The three Panamanian medical physicists, who "tweeked" the software to provide additional capability, were charged with second-degree murder. The U.S. company is faced with serious litigation in two countries. Gage and McCormick comment:

> This is not a cautionary tale for medical technicians, even though they can find themselves fighting to stay out of jail if they misunderstand or misuse technology. This also is not a tale of how human beings can be injured or worse by poorly designed or poorly explained software, although there are plenty of examples to make the point. This is a warning for any creator of computer programs: that software quality matters, that applications must be foolproof, and that—whether embedded in the engine of a car, a robotic arm in a factory or a healing device in a hospital—poorly deployed code can kill.

Poor quality leads to risks, some of them very serious.

14.3.4 Negligence and Liability

The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity. The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., health care administration or homeland security).

Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. Litigation ensues.

In most cases, the customer claims that the developer has been negligent (in the manner in which it has applied software practices) and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.

14.3.5 Quality and Security

As the criticality of Web-based systems and applications grows, application security has become increasingly important. Stated simply, software that does not exhibit high quality is easier to hack, and as a consequence, low-quality software can indirectly increase the security risk with all of its attendant costs and problems.

In an interview in *ComputerWorld,* author and security expert Gary McGraw comments [Wil05]:

> Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware

> of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws.

To build a secure system, you must focus on quality, and that focus must begin during design. The concepts and methods discussed in Part 2 of this book lead to a software architecture that reduces "flaws." By eliminating architectural flaws (thereby improving software quality), you will make it far more difficult to hack the software.

14.3.6 The Impact of Management Actions

Software quality is often influenced as much by management decisions as it is by technology decisions. Even the best software engineering practices can be subverted by poor business decisions and questionable project management actions.

In Part 4 of this book I discuss project management within the context of the software process. As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

Estimation decisions. As I note in Chapter 26, a software team is rarely given the luxury of providing an estimate for a project *before* delivery dates are established and an overall budget is specified. Instead, the team conducts a "sanity check" to ensure that delivery dates and milestones are rational. In many cases there is enormous time-to-market pressure that forces a team to accept unrealistic delivery dates. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.

Scheduling decisions. When a software project schedule is established (Chapter 27), tasks are sequenced based on dependencies. For example, because component **A** depends on processing that occurs within components **B, C,** and **D,** component **A** cannot be scheduled for testing until components **B, C,** and **D** are fully tested. A project schedule would reflect this. But if time is very short, and **A** must be available for further critical testing, you might decide to test **A** without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, **A** may have defects that are hidden, only to be discovered much later. Quality suffers.

Risk-oriented decisions. Risk management (Chapter 28) is one of the key attributes of a successful software project. You really do need to know what might go wrong and establish a contingency plan if it does. Too many software teams prefer blind optimism, establishing a development schedule under the assumption that nothing will go wrong. Worse, they don't have a way of handling things that do go wrong. As a consequence, when a risk becomes a reality, chaos reigns, and as the degree of craziness rises, the level of quality invariably falls.

The software quality dilemma can best be summarized by stating Meskimen's Law—*There's never time to do it right, but always time to do it over again.* My advice: taking the time to do it right is almost never the wrong decision.

14.4 Achieving Software Quality

Software quality doesn't just appear. It is the result of good project management and solid software engineering practice. Management and practice are applied within the context of four broad activities that help a software team achieve high software quality: software engineering methods, project management techniques, quality control actions, and software quality assurance.

14.4.1 Software Engineering Methods

? What do I need to do to affect quality in a positive way?

If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions and factors discussed in Section 14.2.

In Part 2 of this book, I presented a wide array of concepts and methods that can lead to a reasonably complete understanding of the problem and a comprehensive design that establishes a solid foundation for the construction activity. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially.

14.4.2 Project Management Techniques

The impact of poor management decisions on software quality has been discussed in Section 14.3.6. The implications are clear: if (1) a project manager uses estimation to verify that delivery dates are achievable, (2) schedule dependencies are understood and the team resists the temptation to use short cuts, (3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.

In addition, the project plan should include explicit techniques for quality and change management. Techniques that lead to good project management practices are discussed in Part 4 of this book.

14.4.3 Quality Control

? What is software quality control?

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent. Code may be inspected in order to uncover and correct errors before testing commences. A series of testing steps is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals. Quality control activities are discussed in detail throughout the remainder of Part 3 of this book.

WebRef

Useful links to SQA resources can be found at **www.niwotridge.com/Resources/PM-SWEResources/SoftwareQualityAssurance.htm**

14.4.4 Quality Assurance

Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working. Of course, if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues. Software quality assurance is discussed in detail in Chapter 16.

14.5 Summary

Concern for the quality of the software-based systems has grown as software becomes integrated into every aspect of our daily lives. But it is difficult to develop a comprehensive description of software quality. In this chapter quality has been defined as an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

A wide variety of software quality dimensions and factors have been proposed over the years. All try to define a set of characteristics that, if achieved, will lead to high software quality. McCall's and the ISO 9126 quality factors establish characteristics such as reliability, usability, maintainability, functionality, and portability as indicators that quality exists.

Every software organization is faced with the software quality dilemma. In essence, everyone wants to build high-quality systems, but the time and effort required to produce "perfect" software are simply unavailable in a market-driven world. The question becomes, should we build software that is "good enough"? Although many companies do just that, there is a significant downside that must be considered.

Regardless of the approach that is chosen, quality does have a cost that can be discussed in terms of prevention, appraisal, and failure. Prevention costs include all software engineering actions that are designed to prevent defects in the first place. Appraisal costs are associated with those actions that assess software work products to determine their quality. Failure costs encompass the internal price of failure and the external effects that poor quality precipitates.

Software quality is achieved through the application of software engineering methods, solid management practices, and comprehensive quality control—all supported by a software quality assurance infrastructure. In the chapters that follow, quality control and assurance are discussed in some detail.

Problems and Points to Ponder

14.1. Describe how you would assess the quality of a university before applying to it. What factors would be important? Which would be critical?

14.2. Garvin [Gar84] describes five different views of quality. Provide an example of each using one or more well-known electronic products with which you are familiar.

14.3. Using the definition of software quality proposed in Section 14.2, do you think it's possible to create a useful product that provides measurable value without using an effective process? Explain your answer.

14.4. Add two additional questions to each of Garvin's quality dimensions presented in Section 14.2.1.

14.5. McCall's quality factors were developed during the 1970s. Almost every aspect of computing has changed dramatically since the time that they were developed, and yet, McCall's factors continue to apply to modern software. Can you draw any conclusions based on this fact?

14.6. Using the subattributes noted for the ISO 9126 quality factor "maintainability" in Section 14.2.3, develop a set of questions that explore whether or not these attributes are present. Follow the example shown in Section 14.2.4.

14.7. Describe the software quality dilemma in your own words.

14.8. What is "good enough" software? Name a specific company and specific products that you believe were developed using the good enough philosophy.

14.9. Considering each of the four aspects of the cost of quality, which do you think is the most expensive and why?

14.10. Do a Web search and find three other examples of "risks" to the public that can be directly traced to poor software quality. Consider beginning your search at **http://catless.ncl.ac.uk/risks.**

14.11. Are *quality* and *security* the same thing? Explain.

14.12. Explain why it is that many of us continue to live by Meskimen's law. What is it about the software business that causes this?

Further Readings and Information Sources

Basic software quality concepts are considered in books by Henry and Hanlon (*Software Quality Assurance,* Prentice-Hall, 2008), Khan and his colleagues (*Software Quality: Concepts and Practice,* Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality,* Springer, 2002), and Daughtrey (*Fundamental Concepts for the Software Quality Engineer,* ASQ Quality Press, 2001).

Duvall and his colleagues (*Continuous Integration: Improving Software Quality and Reducing Risk,* Addison-Wesley, 2007), Tian (*Software Quality Engineering,* Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices,* Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice,* Alpha Science International, Ltd., 2004), and Galin (*Software Quality Assurance: From Theory to Implementation,* Addison-Wesley, 2003) present detailed treatments of SQA. Quality assurance in the context of the agile process is considered by Stamelos and Sfetsos (*Agile Software Development Quality Assurance,* IGI Global, 2007).

Solid design leads to high software quality. Jayasawal and Patton (*Design for Trustworthy Software,* Prentice-Hall, 2006) and Ploesch (*Contracts, Scenarios and Prototypes,* Springer, 2004) discuss tools and techniques for developing "robust" software.

Measurement is an important component of software quality engineering. Ejiogu (*Software Metrics: The Discipline of Software Quality,* BookSurge Publishing, 2005), Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2002), and Nance and Arthur (*Managing Software Quality,* Springer, 2002) discuss important quality-related metrics and models. The team-oriented aspects of software quality are considered by Evans (*Achieving Software Quality through Teamwork,* Artech House Publishers, 2004).

A wide variety of information sources on software quality is available on the Internet. An up-to-date list of World Wide Web references relevant to software quality can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm.**