2 Defining Quality: What Do You Want?

> Good is not good, where better is expected. ■
>
> *Thomas Fuller,* Church History of Britain, XI.3

So what do we mean by *good* software? In Chapter 1 we saw that the answer is not simple. Indeed, "good" is in the eye of the beholder. Software can be good if it passes its tests without discovery of any new faults. Or it is good if its use makes money, provides a service, or saves time for the consumer who purchased it. Or it is good if it does the same things that the old software did, except more easily. To build or buy solid, survivable software, we need to know what level of quality we seek. The "goodness" or quality of the software depends in part on who is observing and assessing it.

Five Views of Quality

Garvin (1984) has explored how different people perceive quality. He describes quality from five different perspectives:

1. *Transcendental view.* Quality is something we can recognize but not define.
2. *User view.* Quality is fitness for purpose.

3. *Manufacturing view.* Quality is conformance to specification.
4. *Product view.* Quality is tied to inherent product characteristics.
5. *Value-based view.* Quality depends on the amount the customer is willing to pay for it.

The transcendental view is much like Plato's description of the ideal, or Aristotle's concept of form. For example, suppose we want to build a table. We can envision the table we want: the number of legs, the size and position of the top, and the desirable characteristics, such as stability and smoothness of the surface. But just as every actual table is an approximation of an ideal table, we can think of software quality as an ideal toward which we strive. We may never be able to implement its ideal functions and characteristics completely, because of imperfect understanding, imperfect technology, or an imperfect "manufacturing" process. But we still know what we want, and we have a sense of closeness between the actual and ideal products.

The transcendental view is ethereal and one that is not particularly useful to you as a manager. "I'll know it when I see it" does not offer useful governing objectives for organizing and managing a project. By contrast, a user takes a more concrete view of quality. We usually take a user view when we measure software product characteristics, such as availability or reliability, in order to understand the overall product quality. By quantifying the characteristics or identifying recognizable aspects of quality, we can set targets and know when we have met them.

However, viewing product characteristics by themselves paints only a partial picture. We cannot always tell which of our development or maintenance activities is improving the quality as we work. For example, we know the reliability of a software system only after it has been completed and has run for a while. We cannot assess the reliability of its parts and then assume that the whole has a reliability that is a composite of the parts' reliabilities. The same is true for security: A collection of secure parts is not necessarily itself secure. So we need to identify an alternative or additional way of looking at quality that tells us the probable quality as early as possible.

The manufacturing view expands the picture by looking at quality during production and after delivery. In particular, it examines whether the product was built right the first time, avoiding costly rework to fix delivered faults. Thus, the manufacturing view is a process view, advocating conformance to a particular process. However, identifying the right process is an art in itself; most software engineering research addresses issues of matching the right tool or technique with the required product characteristics. Moreover, we often lack convincing evidence that conformance to process actually results

in products with fewer faults and failures; process may indeed lead to high-quality products, but adherence to the wrong process might institutionalize the production of mediocre products. In later chapters we suggest process activities with proven quality enhancements.

While the user and manufacturing views look at the product from the outside, the product view peers inside and evaluates a product's inherent characteristics. For example, users can tell whether or not a product is available for use, but they cannot see whether the code is well structured, whether the inheritance hierarchy is deep or shallow, or whether violations of coding standards are lurking in the product. Developers and maintainers have a privileged, inside view; they can look for faults waiting to become failures. This insider's view of a product is often advocated by software metrics experts, who assume that good internal quality indicators will lead to good external ones. However, more research is needed to verify these assumptions and to determine which aspects of internal quality affect the product's use. For example, consider that there is no convincing evidence that the infamous GOTO statement produces more unreliable products than its avoidance. We can build tools to measure the number of GOTOs and even to tell us where each one resides in the code, but that knowledge does not really contribute to improving product quality. Nevertheless, common wisdom is that GOTOs are bad; this issue sparked a famous controversy some years ago.

Customers or marketers often adopt a user's perspective. Researchers sometimes hold a product view, and the development team usually has a manufacturing view. If the differences in viewpoints are not made explicit, confusion and misunderstanding can lead to bad decisions and poor products. The value-based view can link these disparate pictures of quality. By equating quality with what the customer is willing to pay, we can look at trade-offs between cost and quality, and we can manage conflicts when they arise. Similarly, purchasers can compare product costs with potential benefits, thinking of quality as value for money.

The value-based view highlights the fact that different people value different things. For example, this view helps explain why faults tolerated in word-processing software would not in general be acceptable in safety- or mission-critical systems. Moreover, each of us has a different tolerance for risk as well as a different view of the role of technology in our various businesses. We see this difference most clearly when we examine Microsoft's products. Microsoft balances risk and quality very adroitly, where being first to market trumps taking time to perfect a feature. This conscious decision involves business risk, but it is based on our tolerance as consumers for

problems in the software. If we are uncomfortable with a vendor's quality, we must criticize ourselves as well as the vendor for allowing price or novelty to override our desire for more solid software.

SIDEBAR 2.1
DRIVING YOUR SOFTWARE

Open any major newspaper's business section and you'll find more and more articles about software. These articles are not just about making money in high-technology stocks. They are about how software has become one of the tools with which businesses compete in the marketplace. For example, a recent issue of the *Washington Post* (Brown 2000) describes the importance of software in the automotive industry:

> In the past, Detroit's auto executives competed against one another with muscle, raw horsepower covered by chrome and sheet metal. That has changed. The new competition involves computer chips, electronic sensors, Internet access and other wireless communications, and the ability to exploit those devices and systems to sell more than cars and trucks.…Ford shocked members of the international automotive media with its unveiling of its 24-7 concept vehicles, a linear, box-like trio of cars and hybrid car/trucks laden with telecommunications and electronic infotainment equipment, including devices that could receive wireless Internet feeds of news, email, advertisements and other information.

A Ford executive said that competition is no longer about "varoom and shiny new sheet metal." Instead, he said, "it's about establishing the car as the Internet on wheels." Ford has signed a deal with Yahoo! to provide Internet portal and search services, while GM has made a similar agreement with America Online.

What's happening is that the quality of software and software-based services has pervaded almost every marketplace. So it is becoming more and more clear that software quality is directly related to a business's competitiveness and bottom-line profitability.

Risky Business

Attitudes toward business and technology affect decisions about quality and risk. That is, in deciding what level of quality is acceptable in a product, we balance two ideas: the quality we require and the effects of quality (or lack of it) on our business. Quality demands particular kinds of effort: to build quality in, to assess the quality, to fix any remaining problems, and to maintain a minimum level of quality as the product grows and is transformed over time. But there is a cost associated not only with the quality injection,

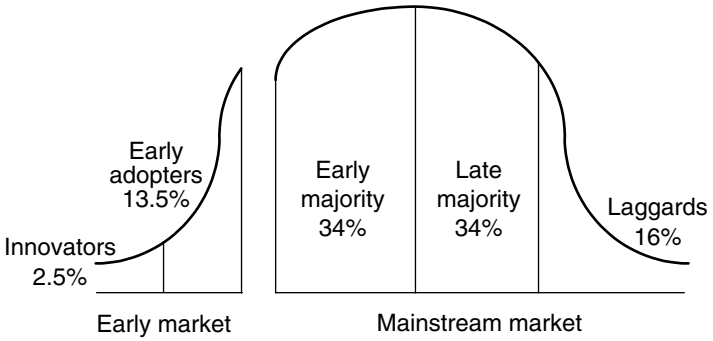

FIGURE 2.1
Adopter types, and the chasm between the early and mainstream markets. [Data from Moore (1991) and Rogers (1995).]

assurance, and maintenance efforts but also with the cost of doing business. It may be cheaper (from the point of view of the developer) to produce a product of suboptimal quality, but it may be expensive from the point of view of the marketplace; customers may not be willing to put up with buggy software, and market share suffers. Or it may be expensive for developers to assure a particular level of quality, but the customers are not willing to pay the price.

Developers implicitly evaluate the tension between quality and business when they decide which technologies (tools, techniques, and processes) to embrace. Rogers (1995) points out that the first people to adopt a technology are innovators; they probably comprise only 2.5 percent of the total likely audience, as shown in Figure 2.1. Innovators are "venturesome"; they are driven by a desire to be rash and to do something daring, and they look to a new technology to implement new functionality. They are willing to take big business and technology risks in order to try something new.

Early adopters are also quick to embrace a new technology. However, it is not just the technology itself that intrigues them. Rather, their interest is driven by their perception of the potential benefits that the new technology can bring to the business. Early adopters usually let someone else test the waters first. But when they read about the success of a technology in a respected publication, they build on someone else's success and introduce the technology to their own organizations. For instance, the Software Technology Support Center (STSC) at Hill Air Force Base (Ogden, Utah) evaluates technology and reports its findings regularly; early adopters may wait for something promising to be reported by the STSC and then embrace those

tools or techniques that sound appealing. By making judicious technology decisions, early adopters decrease the uncertainty about the appropriateness or effectiveness of a new technology.

The early majority is more cautious. Deliberate in decision making, the early majority thinks for some time before welcoming a new technology. Driven by practicality, early majority members make sure that the innovation is not a passing fad. In other words, they follow rather than lead, but they are willing to try new things demonstrated by others to be effective. Early majority adopters can be convinced to try a technology not only when it has succeeded elsewhere but also when it is packaged with materials (such as training guides, help functions, and simple interfaces) that make its use relatively smooth and painless for the developers who will use it. This packaging and support lower the risk of misuse and increase the likelihood that the new technology will make a positive difference to the product's quality.

Late majority adopters are more skeptical. For them, adopting a new technology is usually the result of either economic or peer pressures. Most of the uncertainty about a new idea must be resolved before a late adopter will agree to try it. In other words, a late majority adopter requires substantial evidence that a new technology has made a positive difference to others in the same circumstance. As a result, the late majority will wait until the new technology has become established and there is a sufficient amount of support available. Because late majority adopters dislike uncertainty, they find it particularly appealing to rely on vendor advice. Thus, a vendor can use examples of other customers' experiences to help convince the late majority that the technology will work.

Finally, laggards are often averse to adopting something new, either with economic or personal justification. They jump on the technology bandwagon only when they are certain that a new idea will not fail or when they are forced to change by mandate from managers or customers. Rules imposed by an organization, a standards committee, or a customer can encourage laggards to use a new technology when other methods fail. For example, as we noted in Chapter 1, technology adoption instincts at the Federal Reserve would ordinarily be aligned with laggards. To compete with private industry in the banking world for certain services, the Federal Reserve often pursues technology that it otherwise would ignore until it is "well proven."

Risk and Quality

As a manager, it is important for you to know where you are on the continuum between innovator and laggard. Although no point is inherently bad, understanding your motivation for adopting new techniques and tools can help you decide which new technologies are right for you and your colleagues.

Moore (1991) highlights the chasm, shown in Figure 2.1, between the *early market*, which requires little evidence of a technology's effects on quality, and the *mainstream market*, which requires much more. That is, the mainstream market is more risk-averse than the early market. Because the early market members are innovators and early adopters who are focused on the technology and its capabilities, they are willing to take risks in order to see how the technology works. Moore suggests that the early market is more interested in radical change, whereas the more conservative mainstream is more interested in incremental improvements to an existing way of doing things. That is, the early market takes great leaps and changes the way things are done, whereas the mainstream likes the current process but wants to tinker with it to make it more productive.

As we move from left to right in Figure 2.1, the focus of the technology evaluators changes. The far left is concerned primarily with the technology itself, whereas the far right is most interested in the technology's impact on business. We can make the same distinction by examining technologies along two axes: specialists and generalists, and skeptics and supporters, as shown in Figure 2.2. Innovators tend to be skeptical about a technology, and they evaluate it in the context of their specialized development skills. When gathering information about the technology, they look for very technology-specific evidence: the architecture, schematics, demonstrations, and guru endorsements, for example.

Those with more of a business focus begin to support the technology and look at it in terms of the product it will support. Their evidence comes in the form of benchmarks, product reviews, and demonstrations. As the business becomes still more important, the generalists become more concerned with the marketplace: How much of the market has adopted the technology, and what do industry analysts say? Finally, once generalists are convinced of a technology's value in the marketplace, they look at its impact on the company: What does it do for revenue, and where does it fit into the product line?

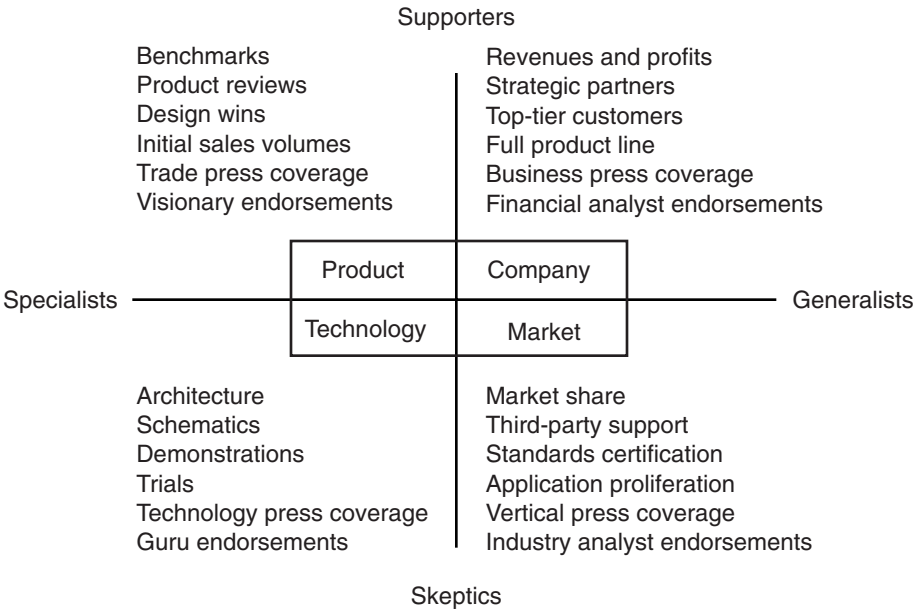

FIGURE 2.2
Evidence to support technology decisions. [After Moore (1991).]

For example, suppose that your organization wants to improve quality by encouraging its developers to use a new design tool. The developers who are skeptical specialists will look at how the tool itself is designed, what platforms it runs on, and what the gurus are saying about it. There is little discussion of how much the tool costs, but many predictions of how many faults and failures will be prevented by using it. On the other hand, those with more of a product focus will examine information about how the tool compares with other design tools, and how much quality is achieved for what price. The market-driven developers will ask very different questions: How much of the design tool market is using this tool? Does this tool adhere to or enforce applicable ISO or IEEE standards? Those with a company focus will determine the effects of the tool on the product line. They want to know how much the tool will cost, how much the product will be improved, and whether the investment is outweighed by the money saved on improved quality.

We can depict these differences by using the notions of technical and business risk, as shown in Figure 2.3. Those with a technology focus are willing to take big technical and business risks; they just want to know how the new technology works. Those with a product focus are a bit more hesitant and

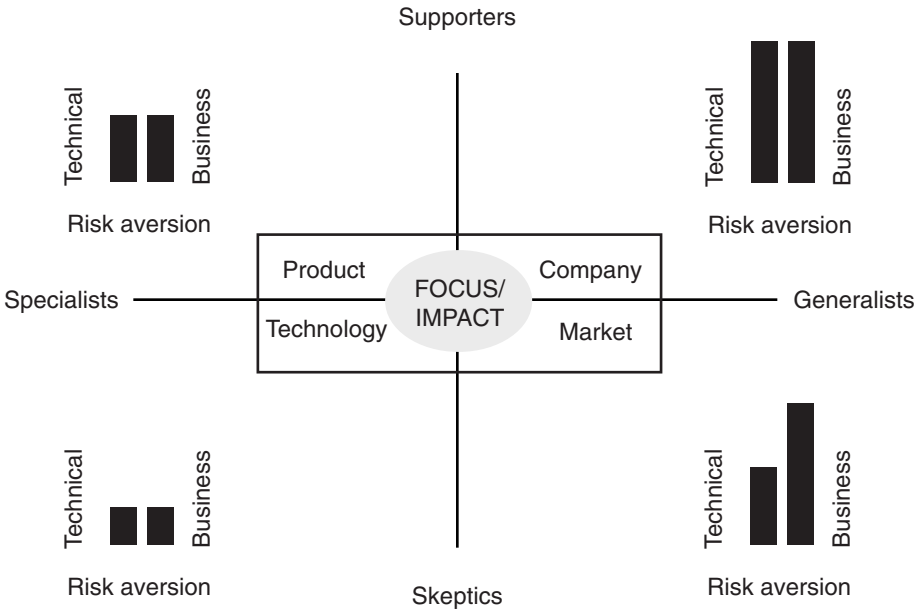

FIGURE 2.3
Risk aversion for different kinds of technologists. [After Moore (1991).]

need a bit more evidence and support (see Sidebar 2.2). When the focus is the marketplace, risk aversion increases unevenly; some technical risk is acceptable, but the technologists are wary of any risk to the business's position in the marketplace. When the focus is the company, any risk, technical or business, threatens the company's financial bottom line.

Consequences of Failure

Ultimately, no matter what our view of quality, each of us wants to prevent our system from failing. The consequences of failure depend in part on our view of quality and on whether the failure involves the process, product, or resources.

Product Failure

When the product fails, its consequences affect us in many ways. The users can lose confidence not only in the product, but also in our ability to fix the current product and to build new products. Even though we may repair faults quickly, the diminished reliability of the system influences our clients'

SIDEBAR 2.2
BUSINESS VALUE VERSUS TECHNICAL VALUE

In a report by Favaro and Pfleeger (1997), Steve Andriole, former chief information officer for Cigna Corporation, a large U.S. insurance company, described how his company distinguished technical value from business value:

> We measure the quality [of our software] by the obvious metrics: up versus down time, maintenance costs, costs connected with modifications, and the like. In other words, we manage development based on operational performance within cost parameters. HOW the vendor provides cost-effective performance is less of a concern than the results of the effort.…The issue of business versus technical value is near and dear to our heart—and one [on] which we focus a great deal of attention. I guess I am surprised to learn that companies would contract with companies for their technical value, at the relative expense of business value. If anything, we err on the other side! If there is not clear (expected) business value (expressed quantitatively: number of claims processed, etc.) then we can't launch a systems project. We take very seriously the "purposeful" requirement phase of the project, when we ask: "why do we want this system?" and "why do we care?"

view of our capabilities. So significant failures, even when they are fixed, can lead to loss in:

- Confidence
- Competitiveness
- Corporate earnings
- Momentum

Let us examine more closely the way in which products can fail. Your organization is likely to have a scale of severity by which you judge the seriousness of a particular failure. Typically, you rate a failure as *minor* if it is an annoyance but can be tolerated for a short time. For example, suppose that a function on a pull-down menu ceases to work but there are function-key work-arounds available until the fix is in place. In this case, the service provided by your system is altered but not halted, so the effect is minor.

On the other hand, a failure is rated *major* if the service ceases for some period of time. You may no longer be able to perform certain functions, resulting in lost time or money. For example, if an airline can no longer issue boarding passes or print flight manifests, the failure is severe.

A failure is *extreme* or *catastrophic* if it involves a significant loss of service or income or endangers the health of one or more people. Leveson and Turner

(1993) describe how the Therac-25 radiation therapy machine malfunctioned, killing several people and injuring several more. Each of these failures may be considered catastrophic. Lions et al. (1996) explain the sequence of events, starting with the reuse of code from *Ariane-4*, which led to the self-destruction of the *Ariane-5* rocket. Sidebar 2.3 presents an example of the severity categories employed by the UK Civil Aviation Authority. Yours may be similar or very different, depending on the nature of your business, your software, and your customers.

SIDEBAR 2.3
UK CIVIL AVIATION AUTHORITY SEVERITY CATEGORIES

Category 1: Operational system critical

- Corruption or loss of data in any stage of processing or presentation, including database
- Inability of any processor, peripheral device, network, or software to meet response times or capacity constraints
- Unintentional failure, halt, or interruption in the operational service of the system or the network for whatever reason
- Failure of any processor or hardware item or any common failure mode point within the quoted mean time between failures that is not returned to service within its mean time to repair

Category 2: System inadequate

- Noncompliance with or omission from the air traffic control operational functions as defined in the functional or system specifications
- Omission in the hardware, software, or documentation (including testing records and configuration management records) as detected in a physical configuration audit
- Any category l item that occurs during acceptance testing of the development and training system configurations

Category 3: System unsatisfactory

- Noncompliance with or omission from the non–air traffic control support and maintenance functions as defined by the functional or system specifications applicable to the operational, development, and training systems
- Noncompliance with standards for deliverable items, including documentation
- Layout or format errors in data presentation that do not affect the operational integrity of the system
- Inconsistency or omission in documentation

The severity of the failure may not be the same as the severity of the outcome from your customer's point of view. For example, you may construct a system that fails frequently but always in a minor way. However, your customers or users may lose confidence in your ability to write or maintain software, because all they remember is that the system always has some problems. Similarly, your software may experience a catastrophic failure, but it happens only once and you fix the problem right away. So your customers remember that the system failed, but their confidence in you is restored slowly as memory of the failure fades (see Sidebar 2.4).

SIDEBAR 2.4
SO HOW GOOD ARE WE?

Hatton (1998) tells us that we can expect commercial software to contain between 6 and 30 faults per 1000 lines of code: a rate that hasn't changed in over 10 years of reporting on project data. The software is considered to be good if it is near the low end, and companies are usually delighted when their code contains fewer than 5 faults per 1000 lines. The U.S. Defense Department presents similar statistics: typically 5 to 15 faults per 1000 lines (*Business Week Online*, December 6, 1999). But we can make a more compelling case for good quality than just trying to keep our faults on the low end of that range. Suppose that your software fails. According to a five-year Defense Department study, it will take you (on average) 1.25 hours to find the source of the failure, and 2 to 9 hours to fix it. So even for "good" code with only 5 faults per 1000 lines, we might need over 50 hours of developer time to clean every 1000 lines—and that doesn't take into account any faults introduced during the correction process. Considering how much you pay your developers and maintainers, the "cost of cleansing" may make your product less competitive or even impossible to sell. So, as we will see in later chapters from both technological and business points of view, it makes sense to take steps early in design and development to eliminate faults and to build in sensible fault handling so that the effects of any failure are mitigated.

A catalog company in the United Kingdom experienced firsthand how a minor technical problem becomes a major customer event. Argos customers were attracted to the $33 price for a Sony Nicam television set posted on the company's Web site on January 8, 2000; hundreds of orders poured in as word spread over the Net about this outrageous price (*Quality Technology Newsletter*, January 9, 2000, *www.qtn.com*). The sets should have been advertised at $3299.00, which is $330 less than their usual price. The erroneous posted price was the result of a round-up error.

Normally, a mistake such as this one would be considered a minor software failure, one easily and quickly correctable. The company refused to honor

any of the orders, and Terry Duddy, Argos chief executive, said: "This was obviously an error we rectified very quickly. We shall be contacting each customer directly to apologize, explaining their orders cannot be accepted in this case." Thus, a minor failure caused a major and expensive headache, not only in terms of bad publicity but also in terms of the cost and time spent to contact disgruntled customers.

Process Failure

Since the 1980s, the software engineering community has looked at process issues to help solve project problems. Using the Process Maturity Model, a collection of Capability Maturity Models, and derivatives from ISO-9000 to SPICE and Trillium, managers have trained the troops to pay careful attention to the way in which the product is specified, designed, implemented, and tested. The idea is this: If your development and maintenance processes are of high quality, so will be the code.

The general notion of "fixing the process" is a good one. The best developers in the world cannot develop high-quality products if they don't take the time to understand the requirements, consider alternative design strategies, assess the impact of a proposed change, or review each other's work. At the same time, there is a tendency for organizations to look for simple solutions to complex problems; some managers find a development process that worked well in one situation and try to impose it on all projects. They use this standard process as a screen to determine whether a particular organization is capable of building solid software. However, Bollinger and McGowan (1991) point out the fatal flaw in this logic: "Just as no government agency would think of using a single test to accredit lawyers, civil engineers and doctors to do government work, it would seem comparably unwise to try to use a single…test to accredit organizations for developing all the many types of application software." Consider, for example, the effects of a hospital which adhered to a well-defined process carried out by untrained medical staff. A good process may be necessary but is certainly not sufficient for a good product. Moreover, we would not expect all software engineers to have the same credentials or experience, just as we do not expect a neurosurgeon and a pediatrician to be able to perform the same kinds of procedures—even when the procedures are clearly defined. That is, every process is embedded in our set of skills, training, and experience.

Similarly, even when we take the care to document a process that has worked well in the past, an organization's documented software development and maintenance processes are not necessarily the ones actually used

by the organization. For example, in their evaluation of the U.S. Department of Defense Software Capability Evaluations, O'Connell and Saiedian (2000) found an elaborate, documented peer review process with four distinct roles: software project manager, auditor, facilitator, and reviewer. The process itself involved three stages: a prereview involving preparation and knowledge gathering; the review itself, involving conflict review and status reporting; and a postreview, where the product is reworked and the quality assessed. But the real review process was much simpler (and far less effective):

1. Find a group of technical people.
2. Give each person a copy of the code.
3. Get opinions and decide whether or not to implement the code.

So if mandated process activities do not work effectively or consistently, how do we improve product quality through process? There are four steps that we can take. First, we can establish quality checkpoints in our processes. That is, we determine where in the process we think we can monitor the quality of the artifacts being received or produced. Then, at each checkpoint, we initiate some activity to assess the quality and determine if development should proceed. For example, suppose we want to be sure that we understand our requirements before we begin design. We may use a requirements review to evaluate the state of the requirements; if some requirements fail the review, we fix them before the design process starts. Or we may use requirements "fit criteria" to make sure that the requirements are sensible and testable before we make them the basis for a design (Robertson and Robertson 1999). On many projects, the requirements change or grow all during development; in this case, we may want to use a configuration management approach, so that all requirements changes are controlled, recorded, and assessed according to their impact on the rest of the system. Thus, the requirements quality activity is determined by the criticality and quality goals of the project. Similarly, we can select activities to evaluate the quality of designs, code, test plans, testing itself, and documentation.

Second, we can integrate process activities with risk analysis. That is, we determine where in the process we are most likely to encounter problems that threaten to cause our products or processes to fail. Then for each high-risk area we devise a process activity to eliminate or mitigate the failure. For example, suppose we are about to build a system that has never before been built. Then the feasibility of the system itself may be at risk. We can introduce prototyping in the process—for aspects of requirements, design, and even testing—to increase the likelihood that we can produce a product that

meets its requirements. Similarly, as we will see in Chapter 3, we can perform a hazard analysis of the design and code to reduce the risk of catastrophic failure.

Third, as we describe in Chapter 6, we must integrate prediction into our process, so that we reduce the number of surprises we will have during development. For example, based on past project history, we can use requirements changes and impact analysis to predict how our resources must be adjusted when the next change is proposed. Similarly, we can use the results of requirements and design reviews to predict how much time we will need to code and test the software and to integrate it with hardware or other systems (see Sidebar 2.5). And we can use the discovery of past faults and failures to predict how much testing we are likely to need. This analysis should reflect the trade-off between features and robustness. If the focus of our development or maintenance process is on the solidity of the software, then resource constraints may prevent us from asking for lots of "filigree" as well. That is, we often find ourselves having to choose between lots of features and lots of quality; we cannot usually have a full measure of both. Prediction can help us find the right balance.

SIDEBAR 2.5
WHY WE CAN'T SEPARATE SOFTWARE PROCESS FROM HARDWARE

When we construct a software process for use on the next project, or when we evaluate a process used on past projects, we often focus too closely on software without acknowledging its role in the larger system (that is, software plus hardware). O'Connell and Saiedian (2000) report an all-too-common incident that shows us why we need to keep our system perspective. They explain that early in the 1990s, the U.S. Department of Defense (DoD) needed a contractor to build a tape reader interface to some tape drive hardware. The DoD awarded a $500,000 contract to the lowest bidder. When the first phase of development was complete, the interface failed testing; the tape reader did not read the tapes consistently. As a result, the project schedule slipped to double its original size, the DoD provided additional funding, and all new development efforts ceased so that the contractor could fix the problem. Eventually, the DoD enlisted the support of an additional organization to try to get the project back on track. Almost immediately, the third party discovered that "the tape drive's read heads were worn out and simply needed to be replaced."

Finally, we must acknowledge the role of human beings in our processes. Leveson (1992) points out that we tend to be "technologically narrow," relying on technical solutions over organizational and managerial solutions. She points out that nearly every major accident at the end of the twentieth cen-

tury (such as Three Mile Island, Chernobyl, and Bhopal) involved serious deficiencies on the parts of both managers and their organizations. As we devise our software development processes, we must assign human responsibility for the result of each activity. As Leveson notes, "management that does not place a high priority on safety can defeat the best efforts of the technical staff."

Resource Failure

The resources used to produce a product—people, components, time, money, and more—can vary a great deal, and the variation can wreak havoc with the result. Solid software requires people with solid skills, but high-quality staff can be difficult to find. For example, the quality of the product often depends on people who understand all of the following:

- The application domain
- How to model and track the requirements
- How to produce a good design, or to choose among competing designs
- The tools used to produce the system
- Programming and design languages and how to write good systems in them
- The nature of the people who will use the system
- Testing
- Maintenance

Similarly, the quality of components requires us to know not only about their functions but also about their vulnerabilities: How do they handle failures? How must they be retested after a change is made?

There are many ways to ensure the quality of our resources. First, we can scrutinize the experience of those proposed to work on a solid software system. It is not enough to ask about years of experience. We need to know if the experience is appropriate to the job at hand. Can the employee understand the application domain? Does the staff know modeling techniques appropriate for generating designs and deciding which design is best? Will the developers be familiar with the set of methods best suited to the tasks that need to be done? If on-the-job training is necessary, will senior developers be available to guide the more junior staff? We must also distinguish between working knowledge and reading knowledge; learning C++ from a textbook or in a classroom is not the same as having used it on a significant project.

As we train the staff to use the tools and techniques chosen, we must also make sure that they understand the tools' and techniques' philosophy. For example, mixing object-oriented design tools with procedural development tools could be a recipe for disaster when the product needs subsequent change. The paradigm mix might make it much more difficult to understand the nature of a change and to maintain the integrity of the design decisions as the product evolves. This situation arises frequently, such as when developers try to optimize their code without understanding the compiler's optimization; as a result, the product is less optimal than it might otherwise have been. It can even be wrong.

Reused components require understanding and scrutiny as well. We must decide whether to test or trust the components. That is, we determine if we are comfortable enough with the components' sources, history, and documentation that we will trust that they perform the desired functions properly. Otherwise, we must devise test data and plans to verify that the components work as advertised. In the latter case, we must determine whether the additional testing for reused components results in more trustworthy code than had we developed the code ourselves, from scratch.

Managers must consider schedule and budget when they think about resources. Although we tend to assume that software estimates and schedules are notoriously optimistic compared with other disciplines, in fact, engineers of all sorts have trouble estimating time and budget, as was amply demonstrated on the Channel tunnel project between England and France. Anyone who has commissioned contractors to upgrade a kitchen or build on a room is unhappily familiar with missed deadlines and extra costs! But software managers are especially prone to optimism when their projects are new or different in some way. Particularly when the projects try to solve problems that have never before been solved (as opposed to automating a known solution), estimation is very difficult. What we want to avoid is what one manager described: The estimators "play with the COCOMO factors" to get an estimate that matches what their "engineering judgment" suggests or what they know the customer wants (in terms of schedule and budget). We suggest three guidelines to help you keep your estimates realistic:

1. *Remember that you cannot accelerate thinking.* Make sure to leave enough time for good requirements analysis and tracking, consideration of design alternatives and trade-offs, and review and change of all artifacts: requirements, design, code, test plans, and documentation.
2. *Produce solid estimates based on historical data as well as expert judgment.* Recent studies of real corporate data reinforce what most managers already know: that expert judgment grounded in project histories

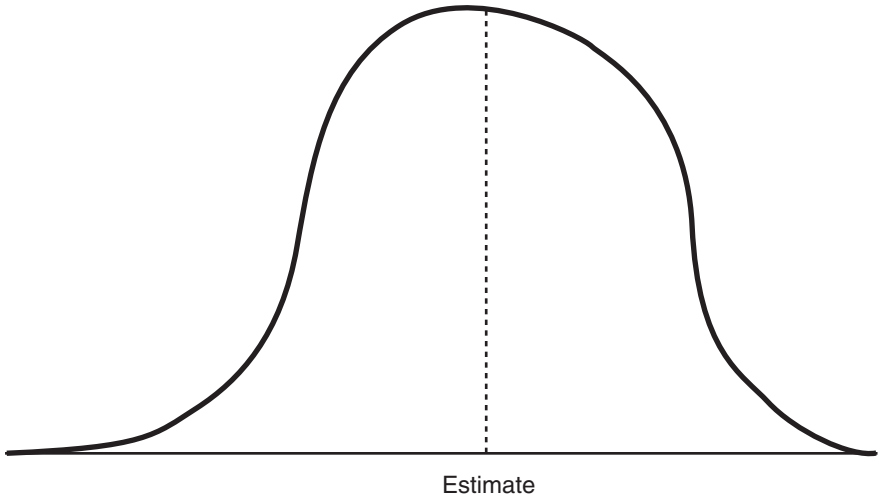

FIGURE 2.4
An estimate is the middle of the distribution. The width or variance of the distribution is also a crucial factor.

yields better estimates than commercial tools or algorithmic techniques. By understanding the key project aspects that take time or require money, and by assessing how current requirements are the same as or different from those aspects on previous projects, you can generate a reasonable picture of how many resources you will need and when during the project they will be needed.

3. *Don't let estimates become targets.* Remember that an estimate is really the center of a distribution, as shown in Figure 2.4. For example, you use project history and expert judgment to determine the least likely value of time needed, the most likely, the maximum time needed, and the way the distribution looks between the two. Your estimate is then the point where 50 percent of the distribution lies to the left and 50 percent to the right. That is, you want your estimate to be right in the middle, so that you have equal probability of being too high as too low. What often happens, though, is that the estimate becomes a target; by eliminating half the distribution, you increase dramatically the chances that you will be wrong, and your "estimate" is almost assuredly below the actual amount needed.

Rules of the Road

So far, we have discussed quality in terms of technology, customer, and market. But there is a fourth component that plays a major role. The business and regulatory environment affects technology trade-offs with respect to critical systems. For example, when we buy a product, we expect the producer to use reasonable care in assuring that the product is safe. As consumers, we assume that food purchased in a reputable grocery or furniture purchased from a reputable department store will not harm us or our environment. We expect the same kind of protection from software, but we do not always get it. Indeed, the warnings on "shrink-wrapped" software often tell us that the developers absolve themselves of any responsibility for its use.

There are laws governing most consumer products, but it is not clear what applies to software. Leveson (1992) points out that if we "do not ourselves insist on establishing minimum levels of competency and safety, then the government will step in and do it for us. The public expects and has the right to expect that dangerous systems are built using the safest technology available."

A number of contemporaneous and conflicting issues have been discussed by a variety of experts (Hatton 1999). So far, many cases that have ended up in the courts have been resolved on contractual issues. However, to be prepared, we should ask ourselves what would be considered negligent conduct in producing software. Until now, negligence has not played a significant role in software-related litigation, but most legal systems tell us when a product does not meet standard requirements for protecting users against unreasonable risk of harm. For example, Voas (2000) points out that to prove negligence, it must be shown that:

- The software or system provider had a legal duty of care owed to the person or business that was injured by the software.
- The duty of care was breached.
- The software was the actual and proximate cause of the injury.
- The person or business was damaged in some way.

The notion of *duty of care* is a positive, active one, in the sense that the party producing the software must take positive steps to ensure that the product is safe. Legal systems also speak of a *standard of care*, meaning that a reasonable person agrees that there are particular actions that most professionals take to ensure the duty of care. For example, we might expect that safety-critical software would be developed with more careful scrutiny of requirements

and design, using inspections and reviews. Someone who produces safety-critical software without having done inspections or reviews might one day be accused of breaching the duty of care.

But there also must be a clear, causal link between the standard of care activities and high-quality software. In other words, if you claim injury (either financial or personal) from the software, you must be able to demonstrate that the lack of duty of care led to the lack of quality that caused your injury. Moreover, according to U.S. and other legal systems, the damage done must be actual, not threatened. That is, you cannot take action based only on the threat of future failure or problems.

In the United States, the National Conference of Commissioners on Uniform State Laws has proposed a law to address consumer concerns about software. Published in July 1999 as an amendment to the Uniform Commercial Code that governs contracts related to the "development, sale, licensing, support and maintenance of computer software and for many other contracts involving automation," the Uniform Computer Information Transactions Act (UCITA) is ambiguous and controversial (Voas 2000). A large and varied collection of organizations, including the Association for Computing Machinery, Consumers Union, and the National Association of Broadcasters, has opposed the amendment, suggesting that it lowers the standards of quality for software.

Whether or not UCITA is passed in its current form where you live (it has already been embraced by several U.S. states), it has unleashed a loud discussion about the nature of software quality and the applicability of current consumer regulations. The conversation grew still louder in February 2000, when Microsoft unveiled its version of Windows 2000 for beta testing, and 63,000 faults were subsequently found. As we saw in Chapter 1, software is not like other products; its acute and unusual sensitivity can have devastating consequences. It remains to be seen how this discrete nature is to be addressed in legislation and regulation. But as software producers and consumers, we must watch carefully as the debate continues. We want to make explicit and deliberate trade-offs about setting and reaching quality goals—and we want to avoid making decisions that are technologically sound but legally, financially, and morally irresponsible or dangerous.

In the following chapters we discuss actions and techniques that might be considered as part of the standard of care. And we present evidence that, by using the methods we describe, you can reduce the risk of negligence, failure, and damage. Sidebar 2.6 covers some things that do not concern us here.

SIDEBAR 2.6
WHAT THIS BOOK IS NOT ABOUT

Software fails in many ways and for many reasons. In this book we explore ways to detect failures and deal with them, and ways to prevent failures from occurring. However, software systems are sometimes perceived to fail because our underlying understanding of the problem to be solved is inadequate or misguided. For example, a recent issue of the *Washington Post* bemoans the failure of the U.S. National Weather Service's software to predict a major storm.

> Instead of the one *inch* of snow forecast by the National Weather Service, we get one to two *feet*. The bad forecast caused the federal government to delay closing, which meant many government workers spent hours struggling to get to work only to learn the government was closing after all. The commute home, in a blizzard, was also hours long for many of them. These bad forecasts can lead us to be trapped in our cars, in airports, on trains. The National Weather Service blamed computers. I thought computers were supposed to help us. (Shaffer 2000)

There are many reasons why weather forecasts are inaccurate, ranging from the uncertainty in knowing the initial state of the atmosphere (as well as the inaccuracy in measuring characteristics of that state) through gross assumptions made in the underlying equations of motion and energy to simple software flaws in the implementation.

Software engineering has come a long way in its short history, and in this book we point out many techniques that can vastly improve the quality of the software we build. But software is a tool for implementing the solution to a problem. If we do not understand the problem, or if our solution is incorrect, inappropriate, or incomplete, all the software engineering in the world will not put the software right. Some of our techniques will highlight the incompleteness of the solution. For instance, hazard analysis will help to point out hazards that have not been addressed properly or at all. But only experts in the application domain can determine if the problem has indeed been solved. Only weather forecasters can improve weather forecasting models; we implement them as best we can. As we have seen, the quality of the model is separate from the quality of the software that implements it.

References

Bollinger, Terry, and Clement McGowan (1991). "A critical look at software capability evaluations." *IEEE Software*, July, pp. 25–41.

Brown, Warren (2000). "Ford, GM drive onto information highway." *Washington Post*, January 15, p. E1.

Favaro, John, and Shari Lawrence Pfleeger (1997). *Making Software Investment Decisions.* Technical report 9701. Washington, DC: Center for Research in Evaluating Software Technology, Howard University, February.

Garvin, David (1984). "What does 'product quality' really mean?" *Sloan Management Review,* Fall, pp. 25–45.

Hatton, Les (1998). "Does OO sync with how we think?" *IEEE Software*, May, pp. 46–54.

——— (1999). "Towards a consistent legal framework for understanding software systems behaviour." LL.M. dissertation. University of Strathclyde Law School, Scotland.

Leveson, Nancy G. (1992). "High-pressure steam engines and computer software." *Proceedings of the International Conference on Software Engineering*, Melbourne, Australia. Los Alamitos, CA: IEEE Computer Society Press, May.

Leveson, Nancy G., and Clark S. Turner (1993). "An investigation of the Therac-25 accidents." *IEEE Computer*, 26(7):18–41.

Lions, J. L., et al. (1996). *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. Paris: European Space Agency. *www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html*.

Moore, Geoffrey (1991). *Crossing the Chasm*. New York: HarperBusiness.

O'Connell, Emilie, and Hossein Saiedian (2000). "Can you trust software capability evaluations?" *IEEE Computer*, 33(2):28–35.

Robertson, James, and Suzanne Robertson (1999). *Mastering the Requirements Process*. Reading, MA: Addison-Wesley.

Rogers, Everett M. (1995). *Diffusion of Innovations*, 4th ed. New York: Free Press.

Shaffer, Ron (2000). "Traffic trouble a certain result of mistaken forecasts." *Washington Post*, January 31, p. B1.

Voas, Jeffrey (2000). "UCITA: take the deal and run." *IT Professional*, January–February, pp. 18–20.