CHAPTER

8 SOFTWARE QUALITY ASSURANCE

KEY CONCEPTS

The software engineering approach described in this book works toward a single goal: to produce high-quality software. Yet many readers will be challenged by the question: "What is software quality?"

Philip Crosby [CRO79], in his landmark book on quality, provides a wry answer to this question:

> The problem of quality management is not what people don't know about it. The problem is what they think they do know . . .
>
> In this regard, quality has much in common with sex. Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! *Software quality assurance* (SQA) is an umbrella activity (Chapter 2) that is applied throughout the software process.

QUICK LOOK

What is it? It's not enough to talk the talk by saying that software quality is important, you have to (1) explicitly define what is meant when you say "software quality," (2) create a set of activities that will help ensure that every software engineering work product exhibits high quality, (3) perform quality assurance activities on every software project, (4) use metrics to develop strategies for improving your software process and, as a consequence, the quality of the end product.

Who does it? Everyone involved in the software engineering 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? Before software quality assurance activities can be initiated, it is important to define 'software quality' at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

What is the work product? A Software Quality Assurance Plan is created to define a software team's SQA strategy. During analysis, design, and code generation, the primary SQA work product is the formal technical review summary report. During ▶

QUICK LOOK

How do I ensure that I've done it right? Find errors before they become defects! That is, work to improve your defect removal efficiency (Chapters 4 and 7), thereby reducing the amount of rework that your software team has to perform.

SQA encompasses (1) a quality management approach, (2) effective software engineering technology (methods and tools), (3) formal technical reviews that are applied throughout the software process, (4) a multitiered testing strategy, (5) control of software documentation and the changes made to it, (6) a procedure to ensure compliance with software development standards (when applicable), and (7) measurement and reporting mechanisms.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does "the right things at the right time in the right way."

8.1 QUALITY CONCEPTS[1]

It has been said that no two snowflakes are alike. Certainly when we watch snow falling it is hard to imagine that snowflakes differ at all, let alone that each flake possesses a unique structure. In order to observe differences between snowflakes, we must examine the specimens closely, perhaps using a magnifying glass. In fact, the closer we look, the more differences we are able to observe.

Quote:

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

This phenomenon, *variation between samples,* applies to all products of human as well as natural creation. For example, if two "identical" circuit boards are examined closely enough, we may observe that the copper pathways on the boards differ slightly in geometry, placement, and thickness. In addition, the location and diameter of the holes drilled in the boards varies as well.

All engineered and manufactured parts exhibit variation. The variation between samples may not be obvious without the aid of precise equipment to measure the geometry, electrical characteristics, or other attributes of the parts. However, with sufficiently sensitive instruments, we will likely come to the conclusion that no two samples of any item are exactly alike.

Variation control is the heart of quality control. A manufacturer wants to minimize the variation among the products that are produced, even when doing something relatively simple like duplicating diskettes. Surely, this cannot be a problem—duplicat-

1 This section, written by Michael Stovsky, has been adapted from "Fundamentals of ISO 9000," a workbook developed for *Essential Software Engineering,* a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

ing diskettes is a trivial manufacturing operation, and we can guarantee that exact duplicates of the software are always created.

Or can we? We need to ensure the tracks are placed on the diskettes within a specified tolerance so that the overwhelming majority of disk drives can read the diskettes. In addition, we need to ensure the magnetic flux for distinguishing a zero from a one is sufficient for read/write heads to detect. The disk duplication machines can, and do, wear and go out of tolerance. So even a "simple" process such as disk duplication may encounter problems due to variation between samples.

KEY POINT

Controlling variation is the key to a high-quality product. In the software context, we strive to control the variation in the process we apply, the resources we expend, and the quality attributes of the end product.

But how does this apply to software work? How might a software development organization need to control variation? From one project to another, we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment, and calendar time. In general, we would like to make sure our testing program covers a known percentage of the software, from one release to another. Not only do we want to minimize the number of defects that are released to the field, we'd like to ensure that the variance in the number of bugs is also minimized from one release to another. (Our customers will likely be upset if the third release of a product has ten times as many defects as the previous release.) We would like to minimize the differences in speed and accuracy of our hotline support responses to customer problems. The list goes on and on.

8.1.1 Quality

The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something." As an attribute of an item, quality refers to measurable characteristics—things we are able to compare to known standards such as length, color, electrical properties, and malleability. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

Nevertheless, measures of a program's characteristics do exist. These properties include cyclomatic complexity, cohesion, number of function points, lines of code, and many others, discussed in Chapters 19 and 24. When we examine an item based on its measurable characteristics, two kinds of quality may be encountered: quality of design and quality of conformance.

Quote:

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

Quality of design refers to the characteristics that designers specify for an item. 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.

Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again, the greater the degree of conformance, the higher is the level of quality of conformance.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused

primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

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 [DEM99] 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.

8.1.2 Quality Control

What is software quality control?

Variation control may be equated to quality control. But how do we achieve quality control? *Quality control* involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. This approach views quality control as part of the manufacturing process.

Quality control activities may be fully automated, entirely manual, or a combination of automated tools and human interaction. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

8.1.3 Quality Assurance

WebRef

A wide variety of software quality resources can be found at **www.qualitytree.com/links/links.htm**

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

8.1.4 Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a base-

line for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms.

What are the components of the cost of quality?

Quality costs may be divided into costs associated with prevention, appraisal, and failure. *Prevention costs* include

- quality planning
- formal technical reviews
- test equipment
- training

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

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

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

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

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
- warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 8.1, based on data collected by Boehm [BOE81] and others, illustrates this phenomenon.

Anecdotal data reported by Kaplan, Clark, and Tang [KAP95] reinforces earlier cost statistics and is based on work at IBM's Rochester development facility:

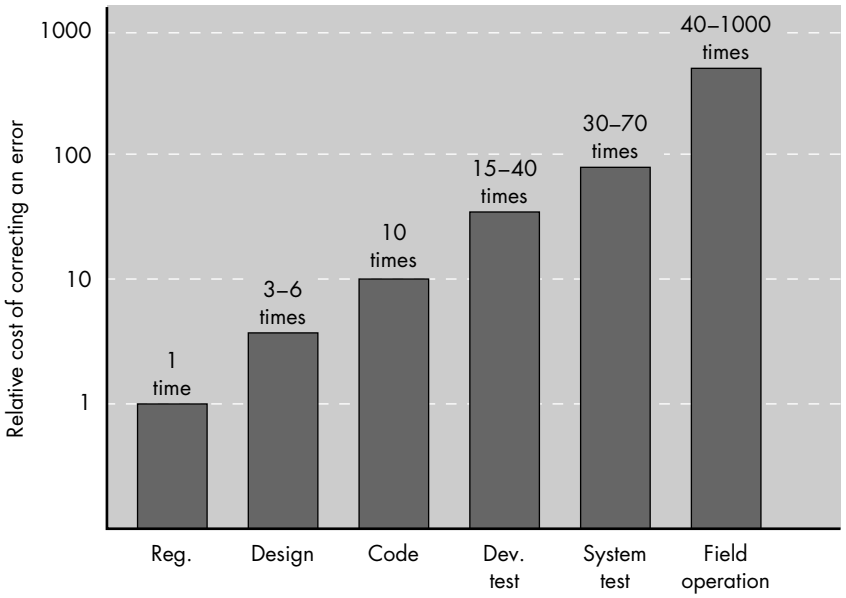

FIGURE 8.1 Relative cost of correcting an error

Testing is necessary, but it's also a very expensive way to find errors. Spend time finding errors early in the process and you may be able to significantly reduce testing and debugging costs.

A total of 7053 hours was spent inspecting 200,000 lines of code with the result that 3112 potential defects were prevented. Assuming a programmer cost of $40.00 per hour, the total cost of preventing 3112 defects was $282,120, or roughly $91.00 per defect.

Compare these numbers to the cost of defect removal once the product has been shipped to the customer. Suppose that there had been no inspections, but that programmers had been extra careful and only one defect per 1000 lines of code [significantly better than industry average] escaped into the shipped product. That would mean that 200 defects would still have to be fixed in the field. At an estimated cost of $25,000 per field fix, the cost would be $5 million, or approximately 18 times more expensive than the total cost of the defect prevention effort.

It is true that IBM produces software that is used by hundreds of thousands of customers and that their costs for field fixes may be higher than those for software organizations that build custom systems. This in no way negates the results just noted. Even if the average software organization has field fix costs that are 25 percent of IBM's (most have no idea what their costs are!), the cost savings associated with quality control and assurance activities are compelling.

8.2 THE QUALITY MOVEMENT

Today, senior managers at companies throughout the industrialized world recognize that high product quality translates to cost savings and an improved bottom line. However, this was not always the case. The quality movement began in the 1940s with the seminal work of W. Edwards Deming [DEM86] and had its first true test in Japan. Using Deming's ideas as a cornerstone, the Japanese developed a systematic

TQM can be applied to computer software. The TQM approach stresses continuous process improvement.

approach to the elimination of the root causes of product defects. Throughout the 1970s and 1980s, their work migrated to the western world and was given names such as "total quality management" (TQM).[2] Although terminology differs across different companies and authors, a basic four step progression is normally encountered and forms the foundation of any good TQM program.

The first step, called *kaizen,* refers to a system of continuous process improvement. The goal of *kaizen* is to develop a process (in this case, the software process) that is visible, repeatable, and measurable.

The second step, invoked only after *kaizen* has been achieved, is called *atarimae hinshitsu.* This step examines intangibles that affect the process and works to optimize their impact on the process. For example, the software process may be affected by high staff turnover, which itself is caused by constant reorganization within a company. Maybe a stable organizational structure could do much to improve the quality of software. *Atarimae hinshitsu* would lead management to suggest changes in the way reorganization occurs.

A wide variety of resources for continuous process improvement and TQM can be found at **deming.eng.clemson.edu/**

While the first two steps focus on the process, the next step, called *kansei* (translated as "the five senses"), concentrates on the user of the product (in this case, software). In essence, by examining the way the user applies the product *kansei* leads to improvement in the product itself and, potentially, to the process that created it.

Finally, a step called *miryokuteki hinshitsu* broadens management concern beyond the immediate product. This is a business-oriented step that looks for opportunity in related areas identified by observing the use of the product in the marketplace. In the software world, *miryokuteki hinshitsu* might be viewed as an attempt to uncover new and profitable products or applications that are an outgrowth from an existing computer-based system.

For most companies *kaizen* should be of immediate concern. Until a mature software process (Chapter 2) has been achieved, there is little point in moving to the next steps.

8.3 SOFTWARE QUALITY ASSURANCE

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? A wag once said, "Every program does something right, it just may not be the thing that we want it to do."

Many definitions of software quality have been proposed in the literature. For our purposes, *software quality* is defined as

How do we define software quality?

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

---

2 See [ART92] for a comprehensive discussion of TQM and its use in a software context and [KAP95] for a discussion of the use of the Baldrige Award criteria in the software world.

There is little question that this definition could be modified or extended. In fact, a definitive definition of software quality could be debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

8.3.1 Background Issues

Quote:

"We made too many wrong mistakes."

Yogi Berra

Quality assurance is an essential activity for any business that produces products to be used by others. Prior to the twentieth century, quality assurance was the sole responsibility of the craftsperson who built a product. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement as key elements of quality management.

Today, every company has mechanisms to ensure quality in its products. In fact, explicit statements of a company's concern for quality have become a marketing ploy during the past few decades.

WebRef

An in-depth tutorial and wide-ranging resources for quality management can be found at **www.management.gov.**

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE94]. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" [SCH98] that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group.

The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors noted in Chapter 19? Has soft-

ware development been conducted according to pre-established standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

8.3.2 SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality assurance and quality control activities) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing. Only reviews are discussed in this chapter. Technology topics are discussed in Parts Three through Five of this book.

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute [PAU93] recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

What is the role of an SQA group?

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- evaluations to be performed
- audits and reviews to be performed
- standards that are applicable to the project
- procedures for error reporting and tracking
- documents to be produced by the SQA group
- amount of feedback provided to the software project team

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Records any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these activities, the SQA group coordinates the control and management of change (Chapter 9) and helps to collect and analyze software metrics.

8.4 SOFTWARE REVIEWS

Like water filters, FTRs tend to retard the "flow" of software engineering activities. Too few and the flow is "dirty." Too many and the flow slows to a trickle. Use metrics to determine which reviews work and which may not be effective. Take the ineffective ones out of the flow.

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.* Freedman and Weinberg [FRE90] discuss the need for reviews this way:

> Technical work needs reviewing for the same reason that pencils need erasers: *To err is human.* The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns:
>
> *O wad some power the giftie give us*
> *to see ourselves as other see us*
>
> A review—any review—is a way of using the diversity of a group of people to:
>
> 1. Point out needed improvements in the product of a single person or team;
>
> 2. Confirm those parts of a product in which improvement is either not desired or not needed;
>
> 3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of

review. In this book, however, we focus on the *formal technical review,* sometimes called a *walkthrough* or an *inspection.* A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

8.4.1 Cost Impact of Software Defects

The *IEEE Standard Dictionary of Electrical and Electronics Terms* (IEEE Standard 100-1992) defines a *defect* as "a product anomaly." The definition for *fault* in the hardware context can be found in IEEE Standard 610.12-1990:

> (a) A defect in a hardware device or component; for example, a short circuit or broken wire. (b) An incorrect step, process, or data definition in a computer program. Note: This definition is used primarily by the fault tolerance discipline. In common usage, the terms "error" and "bug" are used to express this meaning. See also: data-sensitive fault; program-sensitive fault; equivalent faults; fault masking; intermittent fault.

Within the context of the software process, the terms *defect* and *fault* are synonymous. Both imply a quality problem that is discovered *after* the software has been released to end-users (or to another activity in the software process). In earlier chapters, we used the term *error* to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end-user (or to another activity in the software process).

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

The primary objective of an FTR is to find errors before they are passed on to another software engineering activity or released to the customer.

A number of industry studies (by TRW, Nippon Electric, Mitre Corp., among others) indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective [JON86] in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects [IBM81].[3] Assume that an error uncovered during design will cost 1.0 monetary unit to correct. Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units; during testing, 15 units; and after release, between 60 and 100 units.

---

3 Although these data are more than 20 years old, they remain applicable in a modern context.

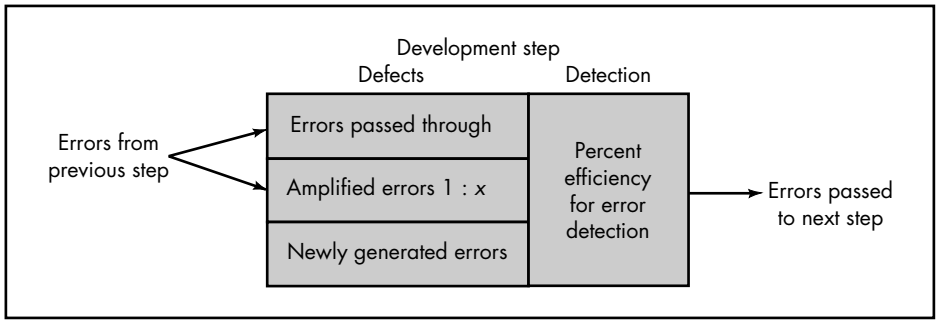

FIGURE 8.2 Defect amplification model

8.4.2 Defect Amplification and Removal

A defect amplification model [IBM81] can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of the software engineering process. The model is illustrated schematically in Figure 8.2. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

Quote:

"Some maladies, as doctors say, at their beginning are easy to cure but difficult to recognize . . . but in the course of time when they have not at first been recognized and treated, become easy to recognize but difficult to cure."

Niccolo Machiavelli

Figure 8.3 illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field. Figure 8.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent errors exist. Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 8.3 and 8.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release). Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

To conduct reviews, a software engineer must expend time and effort and the development organization must spend money. However, the results of the preceding example leave little doubt that we can pay now or pay much more later. Formal tech-

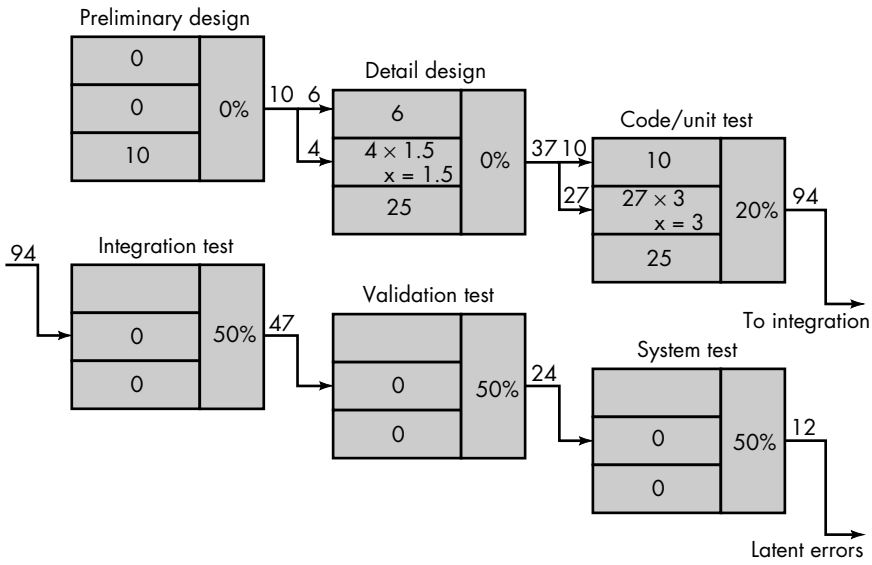

FIGURE 8.3
Defect amplification, no reviews

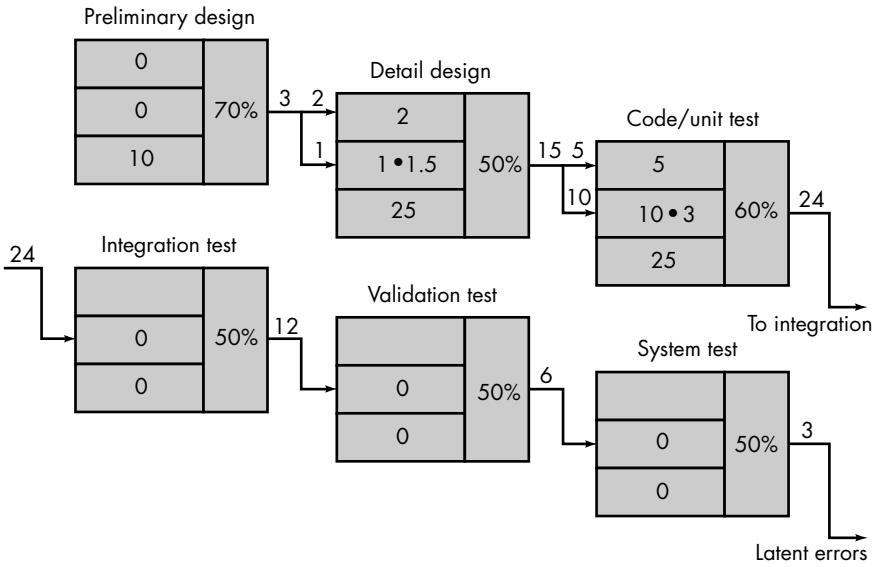

FIGURE 8.4
Defect amplification, reviews conducted

nical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

8.5 FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify

When we conduct FTRs, what are our objectives?

that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR is actually a class of reviews that includes walkthroughs, inspections, round-robin reviews and other small group technical assessments of software. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough [FRE90], [GIL93] are presented as a representative formal technical review.

8.5.1 The Review Meeting

Quote:

"A meeting is too often an event where minutes are taken and hours are wasted."
author unknown

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing focus, the FTR has a higher likelihood of uncovering errors.

KEY POINT

The FTR focuses on a relatively small portion of a work product.

The focus of the FTR is on a work product (e.g., a portion of a requirements specification, a detailed component design, a source code listing for a component). The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required. The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

WebRef

The NASA SATC *Formal Inspection Guidebook* can be downloaded from **satc.gsfc.nasa.gov/fi/fipage.html**

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each.

At the end of the review, all attendees of the FTR must decide whether to (1) accept the product without further modification, (2) reject the product due to severe errors (once corrected, another review must be performed), or (3) accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required). The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

8.5.2 Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A *review summary report* answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

Technical Review Summary Report and Issues List

The review summary report is a single page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The *review issues list* serves two purposes: (1) to identify problem areas within the product and (2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow-up to the review leader.

8.5.3 Review Guidelines

Guidelines for the conduct of formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse that no review at all. The following represents a minimum set of guidelines for formal technical reviews:

Don't point out errors harshly. One way to be gentle is to ask a question that enables the producer to discover his or her own error.

1. *Review the product, not the producer.* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.

2. *Set an agenda and maintain it.* One of the key maladies of meetings of all types is *drift.* An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. *Limit debate and rebuttal.* When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. *Enunciate problem areas, but don't attempt to solve every problem noted.* A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. *Take written notes.* It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. *Limit the number of participants and insist upon advance preparation.* Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).
7. *Develop a checklist for each product that is likely to be reviewed.* A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.
8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as a task during the software engineering process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.
9. *Conduct meaningful training for all reviewers.* To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg [FRE90] estimate a one-month learning curve for every 20 people who are to participate effectively in reviews.
10. *Review your early reviews.* Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

Quote:

"It is one of the most beautiful compensations of life, that no man can sincerely try to help another without helping himself."
Ralph Waldo Emerson

FTR Checklists

Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a

software organization should experiment to determine what approach works best in a local context. Porter and his colleagues [POR95] provide excellent guidance for this type of experimentation.

8.6 FORMAL APPROACHES TO SQA

In the preceding sections, we have argued that software quality is everyone's job; that it can be achieved through competent analysis, design, coding, and testing, as well as through the application of formal technical reviews, a multitiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards. In addition, quality can be defined in terms of a broad array of quality factors and measured (indirectly) using a variety of indices and metrics.

Over the past two decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object [SOM96]. A rigorous syntax and semantics can be defined for every programming language, and work is underway to develop a similarly rigorous approach to the specification of software requirements. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

XRef

Techniques for formal specification of software are considered in Chapter 25. Correctness proofs are considered in Chapter 26.

Attempts to prove programs correct are not new. Dijkstra [DIJ76] and Linger, Mills, and Witt [LIN79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 16).

8.7 STATISTICAL SOFTWARE QUALITY ASSURANCE

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

? What steps are required to perform statistical SQA?

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

Quote:

"20 percent of the code has 80 percent of the defects. Find them, fix them!"

Lowell Arthur

This relatively simple concept represents an important step towards the creation of an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

To illustrate this, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users. Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

- incomplete or erroneous specifications (IES)
- misinterpretation of customer communication (MCC)
- intentional deviation from specifications (IDS)
- violation of programming standards (VPS)
- error in data representation (EDR)
- inconsistent component interface (ICI)
- error in design logic (EDL)
- incomplete or erroneous testing (IET)
- inaccurate or incomplete documentation (IID)
- error in programming language translation of design (PLT)
- ambiguous or inconsistent human/computer interface (HCI)
- miscellaneous (MIS)

WebRef

The Chinese Association for Software Quality presents one of the most comprehensive quality Web sites at **www.casq.org**

To apply statistical SQA, Table 8.1 is built. The table indicates that IES, MCC, and EDR are the *vital few* causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, the software developer might implement facilitated application specification techniques (Chapter 11) to improve the quality of customer communication and specifications. To improve EDR, the developer might acquire CASE tools for data modeling and perform more stringent data design reviews.

It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement [ART97]. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

In conjunction with the collection of defect information, software developers can calculate an *error index* (EI) for each major step in the software process [IEE94]. After analysis, design, coding, testing, and release, the following data are gathered:

E_i = the total number of errors uncovered during the ith step in the software engineering process

TABLE 8.1 DATA COLLECTION FOR STATISTICAL SQA

	Total		Serious		Moderate		Minor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

S_i = the number of serious errors
M_i = the number of moderate errors
T_i = the number of minor errors
PS = size of the product (LOC, design statements, pages of documentation) at the ith step
w_s, w_m, w_t = weighting factors for serious, moderate, and trivial errors, where recommended values are $w_s = 10$, $w_m = 3$, $w_t = 1$. The weighting factors for each phase should become larger as development progresses. This rewards an organization that finds errors early.

At each step in the software process, a *phase index,* PI_i, is computed:

$$PI_i = w_s (S_i/E_i) + w_m (M_i/E_i) + w_t (T_i/E_i)$$

The *error index* is computed by calculating the cumulative effect on each PI_i, weighting errors encountered later in the software engineering process more heavily than those encountered earlier:

$$EI = \Sigma(i \times PI_i)/PS$$
$$= (PI_1 + 2PI_2 + 3PI_3 + \ldots iPI_i)/PS$$

The error index can be used in conjunction with information collected in Table 8.1 to develop an overall indication of improvement in software quality.

The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

A comprehensive discussion of statistical SQA is beyond the scope of this book. Interested readers should see [SCH98], [KAP95], or [KAN95].

8.8 SOFTWARE RELIABILITY

There is no doubt that the reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are acceptable.

The Reliability Analysis Center provides much useful information on reliability, maintainability, supportability, and quality at **rac.iitri.org**

Software reliability, unlike many other quality factors, can be measured directed and estimated using historical and developmental data. *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [MUS87]. To illustrate, program X is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program X were to be executed 100 times and require eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 96 times out of 100.

Whenever software reliability is discussed, a pivotal question arises: What is meant by the term *failure?* In the context of any discussion of software quality and reliability, failure is nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

8.8.1 Measures of Reliability and Availability

Software reliability problems can almost always be traced to errors in design or implementation.

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory (e.g., [ALV64]) to the prediction of software reliability. Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear (see Chapter 1) does not enter into the picture.

There has been debate over the relationship between key concepts in hardware reliability and their applicability to software (e.g., [LIT89], [ROO90]). Although an irrefutable link has yet be be established, it is worthwhile to consider a few simple concepts that apply to both system elements.

If we consider a computer-based system, a simple measure of reliability is *mean-time-between-failure* (MTBF), where

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

? Why is MTBF a more useful metric than defects/KLOC?

Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP. Stated simply, an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the same failure rate, the total error count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 14 months. Many errors in this program may remain undetected for decades before they are discovered. The MTBF of such obscure errors might be 50 or even 100 years. Other errors, as yet undiscovered, might have a failure rate of 18 or 24 months. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

$$\text{Availability} = [\text{MTTF}/(\text{MTTF} + \text{MTTR})] \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

8.8.2 Software Safety

Leveson [LEV86] discusses the impact of software in safety critical systems when she writes:

> Before software was used in safety critical systems, they were often controlled by conventional (nonprogrammable) mechanical and electronic devices. System safety techniques are designed to cope with random failures in these [nonprogrammable] systems. Human design errors are not considered since it is assumed that all faults caused by human errors can be avoided completely or removed prior to delivery and operation.

Quote:

"I cannot imagine any condition which would cause this ship to founder. Modern shipbuilding has gone beyond that."

E.I. Smith, captain of the Titanic

When software is used as part of the control system, complexity can increase by an order of magnitude or more. Subtle design faults induced by human error—something that can be uncovered and eliminated in hardware-based conventional control—become much more difficult to uncover when software is used.

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be

- causes uncontrolled acceleration that cannot be stopped
- does not respond to depression of brake pedal (by turning off)

- does not engage when switch is activated
- slowly loses or gains speed

Worthwhile papers on software safety (and a detailed glossary) can be found at **www.rstcorp.com/hotlist/topics-safety.html**

Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.[4] To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as *fault tree analysis* [VES81], *real-time logic* [JAN86], or *petri net models* [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events. The role of software in managing undesirable events is then indicated.

? What is the difference between software reliability and software safety?

Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them. Software reliability uses statistical analysis to determine the likelihood that a software failure will occur. However, the occurrence of a failure does not necessarily result in a hazard or mishap. Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is, failures are not considered in a vacuum, but are evaluated in the context of an entire computer-based system.

A comprehensive discussion of software safety is beyond the scope of this book. Those readers with further interest should refer to Leveson's [LEV95] book on the subject.

8.9 MISTAKE-PROOFING FOR SOFTWARE

If William Shakespeare had commented on the modern software engineer's condition, he might have written: "To err is human, to find the error quickly and correct it is divine." In the 1960s, a Japanese industrial engineer, Shigeo Shingo [SHI86], working at Toyota, developed a quality assurance technique that led to the prevention and/or early correction of errors in the manufacturing process. Called *poka-yoke* (mistake-proofing), Shingo's concept makes use of *poka-yoke* devices—mechanisms that lead to (1) the prevention of a potential quality problem before it occurs or (2) the rapid detection of quality problems if they are introduced. We encounter *poka-yoke* devices in our everyday lives (even if we are unaware of the concept). For exam-

4 This approach is analogous to the risk analysis approach described for software project management in Chapter 6. The primary difference is the emphasis on technology issues as opposed to project-related topics.

ple, the ignition switch for an automobile will not work if an automatic transmission is in gear (a *prevention device*); an auto's warning beep will sound if the seat belts are not buckled (a *detection device*).

A comprehensive collection of *poka-yoke* resources can be obtained at **www.campbell.berry.edu/faculty/jgrout/pokayoke.shtml**

An effective *poka-yoke* device exhibits a set of common characteristics:

- **It is simple and cheap.** If a device is too complicated or expensive, it will not be cost effective.
- **It is part of the process.** That is, the *poka-yoke* device is integrated into an engineering activity.
- **It is located near the process task where the mistakes occur.** Thus, it provides rapid feedback and error correction.

Although *poka-yoke* was originally developed for use in "zero quality control" [SHI86] for manufactured hardware, it can be adapted for use in software engineering. To illustrate, we consider the following problem [ROB97]:

> A software products company sells application software to an international market. The pull-down menus and associated mnemonics provided with each application must reflect the local language. For example, the English language menu item for "Close" has the mnemonic "C" associated with it. When the application is sold in a French-speaking country, the same menu item is "Fermer" with the mnemonic "F." To implement the appropriate menu entry for each locale, a "localizer" (a person conversant in the local language and terminology) translates the menus accordingly. The problem is to ensure that (1) each menu entry (there can be hundreds) conforms to appropriate standards and that there are no conflicts, regardless of the language that is used.

The use of *poka-yoke* for testing various application menus implemented in different languages as just described is discussed in a paper by Harry Robinson [ROB97]:[5]

> We first decided to break the menu testing problem down into parts that we could solve. Our first advance on the problem was to understand that there were two separate aspects to the message catalogs. There was the content aspect: the simple text translations, such as changing "Close" to "Fermer." Since the test team was not fluent in the 11 target languages, we had to leave this aspect to the language experts.
>
> The second aspect of the message catalogs was the structure, the syntax rules that a properly constructed target catalog must obey. Unlike content, it would be possible for the test team to verify the structural aspects of the catalogs.
>
> As an example of what is meant by structure, consider the labels and mnemonics of an application menu. A menu is made up of labels and associated mnemonics. Each menu, regardless of its contents or its locale, must obey the following rules listed in the Motif Style Guide:
>
> - Each mnemonic must be contained in its associated label
> - Each mnemonic must be unique within the menu

5 The paragraphs that follow have been excerpted (with minor editing) from [ROB97] with the permission of the author.

- Each mnemonic must be a single character
- Each mnemonic must be in ASCII

These rules are invariant across locales, and can be used to verify that a menu is constructed correctly in the target locale.

There were several possibilities for how to mistake-proof the menu mnemonics:

Prevention device. We could write a program to generate mnemonics automatically, given a list of the labels in each menu. This approach would prevent mistakes, but the problem of choosing a good mnemonic is difficult and the effort required to write the program would not be justified by the benefit gained.

Prevention device. We could write a program that would prevent the localizer from choosing mnemonics that did not meet the criteria. This approach would also prevent mistakes, but the benefit gained would be minimal; incorrect mnemonics are easy enough to detect and correct after they occur.

Detection device. We could provide a program to verify that the chosen menu labels and mnemonics meet the criteria above. Our localizers could run the programs on their translated message catalogs before sending the catalogs to us. This approach would provide very quick feedback on mistakes, and it is likely as a future step.

Detection device. We could write a program to verify the menu labels and mnemonics, and run the program on message catalogs after they are returned to us by the localizers. This approach is the path we are currently taking. It is not as efficient as some of the above methods, and it can require communication back and forth with the localizers, but the detected errors are still easy to correct at this point.

Several small poka-yoke scripts were used as poka-yoke devices to validate the structural aspects of the menus. A small poka-yoke script would read the table, retrieve the mnemonics and labels from the message catalog, and compare the retrieved strings against the established criteria noted above.

The poka-yoke scripts were small (roughly 100 lines), easy to write (some were written in under an hour) and easy to run. We ran our poka-yoke scripts against 16 applications in the default English locale plus 11 foreign locales. Each locale contained 100 menus, for a total of 1200 menus. The poka-yoke devices found 311 mistakes in menus and mnemonics. Few of the problems we uncovered were earth-shattering, but in total they would have amounted to a large annoyance in testing and running our localized applications.

This example depicts a *poka-yoke* device that has been integrated into software engineering testing activity. The *poka-yoke* technique can be applied at the design, code, and testing levels and provides an effective quality assurance filter.

8.10 THE ISO 9000 QUALITY STANDARDS[6]

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

6 This section, written by Michael Stovsky, has been adapted from "Fundamentals of ISO 9000" and "ISO 9001 Standard," workbooks developed for *Essential Software Engineering,* a video curriculum developed by R. S. Pressman & Associates, Inc. Reprinted with permission.

[ANS87]. Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

Extensive links to ISO 9000/9001 resources can be found at **www.tantara.ab.ca/iso_list.htm**

The ISO 9000 standards have been adopted by many countries including all members of the European Community, Canada, Mexico, the United States, Australia, New Zealand, and the Pacific Rim. Countries in Latin and South America have also shown interest in the standards.

After adopting the standards, a country typically permits only ISO registered companies to supply goods and services to government agencies and public utilities. Telecommunication equipment and medical devices are examples of product categories that must be supplied by ISO registered companies. In turn, manufacturers of these products often require their suppliers to become registered. Private companies such as automobile and computer manufacturers frequently require their suppliers to be ISO registered as well.

To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semi-annual surveillance audits ensure continued compliance to the standard.

8.10.1 The ISO Approach to Quality Assurance Systems

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO compliant, these processes must address the areas identified in the standard and must be documented and practiced as described.

ISO 9000 describes what must be done to be compliant, but it does not describe how it must be done.

ISO 9000 describes the elements of a quality assurance system in general terms. These elements include the organizational structure, procedures, processes, and resources needed to implement quality planning, quality control, quality assurance, and quality improvement. However, ISO 9000 does not describe how an organization should implement these quality system elements. Consequently, the challenge lies in designing and implementing a quality assurance system that meets the standard and fits the company's products, services, and culture.

8.10.2 The ISO 9001 Standard

ISO 9001 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001 standard is applicable to all engineering

disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interpret the standard for use in the software process.

ISO 9000 for Software

The requirements delineated by ISO 9001 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. For further information on ISO 9001, the interested reader should see [HOY98], [SCH97], or [SCH94].

8.11 THE SQA PLAN

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities that are instituted for each software project.

The SQA plan

A standard for SQA plans has been recommended by the IEEE [IEE94]. Initial sections describe the purpose and scope of the document and indicate those software process activities that are covered by quality assurance. All documents noted in the *SQA Plan* are listed and all applicable standards are noted. The management section of the plan describes SQA's place in the organizational structure, SQA tasks and activities and their placement throughout the software process, and the organizational roles and responsibilities relative to product quality.

The documentation section describes (by reference) each of the work products produced as part of the software process. These include

- project documents (e.g., project plan)
- models (e.g., ERDs, class hierarchies)
- technical documents (e.g., specifications, test plans)
- user documents (e.g., help files)

In addition, this section defines the minimum set of work products that are acceptable to achieve high quality.

The standards, practices, and conventions section lists all applicable standards and practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines). In addition, all project, process, and (in some instances) product metrics that are to be collected as part of software engineering work are listed.

The reviews and audits section of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group, and the customer. It provides an overview of the approach for each review and audit.

The test section references the *Software Test Plan and Procedure* (Chapter 18). It also defines test record-keeping requirements. Problem reporting and corrective action defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities.

The remainder of the *SQA Plan* identifies the tools and methods that support SQA activities and tasks; references software configuration management procedures for controlling change; defines a contract management approach; establishes methods for assembling, safeguarding, and maintaining all records; identifies training required to meet the needs of the plan; and defines methods for identifying, assessing, monitoring, and controlling risk.

8.12 SUMMARY

Software quality assurance is an umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, *poka-yoke* devices, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

SQA is complicated by the complex nature of software quality—an attribute of computer programs that is defined as "conformance to explicitly and implicitly specified requirements." But when considered more generally, software quality encompasses many different product and process factors and related metrics.

Software reviews are one of the most important SQA activities. Reviews serve as filters throughout all software engineering activities, removing errors while they are relatively inexpensive to find and correct. The formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors.

To properly conduct software quality assurance, data about the software engineering process should be collected, evaluated, and disseminated. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

In summary, we recall the words of Dunn and Ullman [DUN82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.