SOFTWARE ENGINEERING

UNIT V

1. Quality Concepts: 2.Software Quality – 3.Achieving Software Quality – 4.Formal Technical Reviews – 5.Software Quality Assurance: 6.Elements of SQA – 7.SQA Tasks, Goals and Metrics – 8.Statistical SQA – 9.Software Reliability – 10.Process and Project Metrics: 11.Metrics in the Process and Project Domains – 12.Software Measurement – 13.Metrics for Software Quality – 14.Establishing a Software Metrics Program.

1. QUALITY CONCEPTS

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

Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn’t include the cost of losing angry customers. Because IT shops write applications that rely on packaged infrastructure software, bad code can wreak havoc on custom apps as well. .

Just how bad is bad software? Definitions vary, but experts say it takes only three or four defects per 1,000 lines of code to make a program perform poorly. Factor in that most programmers inject about one error for every 10 lines of code they write, multiply that by the millions of lines of code in many commercial products, then figure it costs software vendors at least half their development budgets to fix errors while testing. Get the picture?

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

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

**SOFTWARE QUALITY**

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

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

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

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications

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

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

**User Satisfaction = Compliant Product + Good Quality + Delivery within Budget and Schedule**

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

**2. SOFTWARE QUALITY**

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define software quality? In the most general sense, software quality can be defined1 as: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

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

**1.** **An effective software process** establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high-quality software. Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

**2.** **A useful product** delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software.

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

**Garvin’s Quality Dimensions**

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

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

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

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

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

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

**Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period? Can support staff acquire all information they need to make changes or correct defects? Douglas Adams [Ada93] makes a wry comment that seems appropriate here: “The difference between something that can go wrong and something that can’t possibly go wrong is that when something that can’t possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.”

**Aesthetics.** There’s no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious “presence” that are hard to quantify

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

Garvin’s quality dimensions provide you with a “soft” look at software quality. Many (but not all) of these dimensions can only be considered subjectively. For this reason, you also need a set of “hard” quality factors that can be categorized in two broad groups:

(1) Factors that can be directly measured (e.g., defects uncovered during testing)

(2) Factors that can be measured only indirectly (e.g., usability or maintainability).

In each case measurement must occur. You should compare the software to some datum and arrive at an indication of quality.

**McCall’s Quality Factors**

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

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

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

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

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

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

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

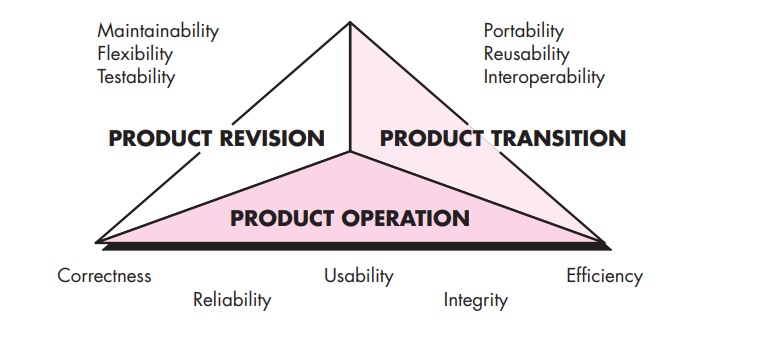
**Maintainability**. Effort required to locate and fix an error in a program. [This is a very limited definition.]

**Flexibility**. Effort required to modify an operational program.

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

**Portability**. Effort required to transfer the program from one hardware and/or software system environment to another.   
**Reusability**. Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

**Interoperability**. Effort required to couple one system to another

****

It is difficult, and in some cases impossible, to develop direct measures2 of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality.

**ISO 9126 Quality Factors**

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

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

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

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

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

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

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

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

**Targeted Quality Factors**

The quality dimensions and factors presented in Sections 14.2.1 and 14.2.2 focus on the software as a whole and can be used as a generic indication of the quality of an application. A software team can develop a set of quality characteristics and associated questions that would probe3 the degree to which each factor has been satisfied. For example, McCall identifies usability as an important quality factor. If you were asked to review a user interface and assess its usability, how would you proceed? You might start with the subattributes suggested by McCall—understandability, learnability, and operability—but what do these mean in a pragmatic sense?

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

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

• Is the interface layout conducive to easy understanding?

• Are interface operations easy to locate and initiate?

• Does the interface use a recognizable metaphor?

• Is input specified to economize key strokes or mouse clicks?

• Does the interface follow the three golden rules? (Chapter 11)

• Do aesthetics aid in understanding and usage?

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

• Does the interface layout and style allow a user to locate operations and information efficiently?

• Can a sequence of operations (or data input) be performed with an economy of motion?

• Are output data or content presented so that it is understood immediately?

• Have hierarchical operations been organized in a way that minimizes the depth to which a user must navigate to get something done?

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

• Will the software recognize the error if data at or just outside prescribed boundaries is input? More importantly, will the software continue to operate without failure or degradation?

• Will the interface recognize common cognitive or manipulative mistakes and explicitly guide the user back on the right track?

• Does the interface provide useful diagnosis and guidance when an error condition (associated with software functionality) is uncovered?

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

• Can the interface be customized to the specific needs of a user?

• Does the interface provide a macro capability that enables a user to identify a sequence of common operations with a single action or command?

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

**3. Achieving Software Quality**

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

1. **Software Engineering Methods,**
2. **Project Management Techniques,**
3. **Quality Control Actions,**
4. **Software Quality Assurance.**

**Software Engineering Methods**

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

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

**Project Management Techniques**

The impact of poor management decisions on software quality has been discussed in Section 14.3.6. The implications are clear: if

(1) a project manager uses estimation to verify that delivery dates are achievable,

(2) schedule dependencies are understood and the team resists the temptation to use short cuts,

(3) risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.

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

**Quality Control**

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

**Quality Assurance**

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

**4. Formal Technical Reviews**

A formal technical review (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:

(1) to uncover errors in function, logic, or implementation for any representation of the software;

(2) to verify 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;

(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 and inspections. 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 are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs, see [Rad02], [Wie02], or [Fre90].

1. **The Review Meeting**
2. **Review Reporting and Record Keeping**
3. **Review Guidelines**
4. **Sample-Driven Reviews**

**The Review Meeting**

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 the focus, the FTR has a higher likelihood of uncovering errors.

The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code 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.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of a 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). After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team’s findings.

**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?

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

You should 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.

**Review Guidelines**

Guidelines for conducting 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 than no review at all. The following represents a minimum set of guidelines for formal technical reviews:

**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. Alternatively, notes may be entered directly into a notebook computer.

**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 testing work products.

**8. Allocate resources and schedule time for FTRs.** For reviews to be effective, they should be scheduled as tasks during the software 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.

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

**Sample-Driven Reviews**

In an ideal setting, every software engineering work product would undergo a formal technical review. In the real word of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized.

Thelin and his colleagues [The01] suggest a sample-driven review process in which samples of all software engineering work products are inspected to determine which work products are most error prone. Full FTR resources are then focused only on those work products that are likely (based on data collected during sampling) to be error prone.

To be effective, the sample-driven review process must attempt to quantify those work products that are primary targets for full FTRs. To accomplish this, the following steps are suggested [The01]:

1. Inspect a fraction ai of each software work product i. Record the number of faults fi found within ai .

2. Develop a gross estimate of the number of faults within work product i by multiplying fi by 1/ai .

3. Sort the work products in descending order according to the gross estimate of the number of faults in each.

4. Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must be representative of the work product as a whole and large enough to be meaningful to the reviewers who do the sampling. As ai increases, the likelihood that the sample is a valid representation of the work product also increases. However, the resources required to do sampling also increase. A software engineering team must establish the best value for ai for the types of work products produced.

**5. Software Quality Assurance:**

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 (often called quality management) is an umbrella activity (Chapter 2) that is applied throughout the software process.

Software quality assurance (SQA) encompasses

(1) An SQA process,

(2) Specific quality assurance and quality control tasks (including technical reviews and a multi-tiered testing strategy),

(3) Effective software engineering practice (methods and tools),

(4) Control of all software work products and the changes made to them (Chapter 22),

(5) A procedure to ensure compliance with software development standards (when applicable),

(6) Measurement and reporting mechanisms.

**6. Elements of SQA**

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner [Hor03]:

1. **Standards**
2. **Reviews and audits**
3. **Testing**
4. **Error/defect collection and analysis.**
5. **Change management**
6. **Education**
7. **Vendor Management**
8. **Security management**
9. **Safety**
10. **Risk Management**

**Standards.** The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

**Reviews and audits.** Technical reviews are a quality control activity performed by software engineers for software engineers (Chapter 15). Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

**Testing.** Software testing (Chapters 17 through 20) is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

**Error/defect collection and analysis.** The only way to improve is to measure how you’re doing. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

**Change management.** Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices (Chapter 22) have been instituted.

**Education.** Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement (Chapter 30) and is a key proponent and sponsor of educational programs.

**Vendor management.** Three categories of software are acquired from external software vendors—shrink-wrapped packages (e.g., Microsoft Office), a tailored shell [Hor03] that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and contracted software that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.

**Security management.** With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security.

**Safety.** Because software is almost always a pivotal component of humanrated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

**Risk management.** Although the analysis and mitigation of risk (Chapter 28) is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

In addition to each of these concerns and activities, SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern.

**7. SQA Tasks, Goals and Metrics**

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 control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

**SQA Tasks**

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

1. **Prepares an SQA plan for a project**
2. **Participates in the development of the project’s software process description.**
3. **Reviews software engineering activities to verify compliance with the defined software process**
4. **Audits designated software work products to verify compliance with those defined as part of the software process**
5. **Ensures that deviations in software work and work products are documented and handled according to a documented procedure.**
6. **Records any noncompliance and reports to senior management.**

**Prepares an SQA plan for a project.** The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance actions 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 conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software 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 software engineering work products.

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

In addition to these actions, the SQA group coordinates the control and management of change (Chapter 22) and helps to collect and analyze software metrics.

**Goals, Attributes, and Metrics**

The SQA actions described in the preceding section are performed to achieve a set of pragmatic goals:

1. **Requirements quality**
2. **Design quality**
3. **Code quality**
4. **Quality control effectiveness**

**Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

**Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality**.**

**Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

**Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

Figure 16.1 (adapted from [Hya96]) identifies the attributes that are indicators for the existence of quality for each of the goals discussed. Metrics that can be used to indicate the relative strength of an attribute are also shown.

|  |  |  |
| --- | --- | --- |
| **Goal** | **Attribute** | **Metrics** |
| **Requirement Quality** | Ambiguity | Number of ambiguous modifiers (e.g., many, large, human-friendly) |
| Completeness | Number of TBA, TBD |
| Understandability | Number of sections/subsections |
| Volatility | Number of changes per requirement  Time (by activity) when change is requested |
| Traceability | Number of requirements not traceable to design/code |
| Model clarity | Number of UML models Number of descriptive pages per model  Number of UML errors |
| **Design Quality** | Architectural integrity | Existence of architectural model |
| Component completeness | Number of components that trace to architectural model Complexity of procedural design |
| Interface complexity | Average number of pick to get to a typical function or content Layout appropriateness |
| Patterns | Number of patterns used |
| **Goals** | **Attributes** | **Metrics** |
| **Code Quality** | Complexity | Cyclomatic complexity |
| Maintainability | Design factors |
| Understandability | Percent internal comments Variable naming conventions |
| Reusability | Percent reused components |
| Documentation | Readability index |
| **QC Effectiveness** | Resource allocation | Staff hour percentage per activity |
| Completion rate | Actual vs. budgeted completion time |
| Review effectiveness | review metrics |
| Testing effectiveness | Number of errors found and criticality  Effort required to correct an error Origin of error |

**8. Statistical SQA**

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

1. Information about software errors and defects is collected and categorized.

2. An attempt is made to trace each error and defect to its underlying cause (e.g., non - conformance 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 errors and defects.

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

**A Generic Example**

To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on errors and defects for a period of one year. Some of the errors are uncovered as software is being developed. Others (defects) are encountered after the software has been released to its end users. Although hundreds of different problems 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)

To apply statistical SQA, the table in Figure 16.2 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, you might implement requirements gathering techniques (Chapter 5) to improve the quality of customer communication and specifications. To improve EDR, you might acquire 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

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!

**Six Sigma for Software Engineering**

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy “is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company’s operational performance by identifying and eliminating defects’ in manufacturing and service-related processes” [ISI08]. The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

• Define customer requirements and deliverables and project goals via welldefined methods of customer communication.

• Measure the existing process and its output to determine current quality performance (collect defect metrics).

• Analyze defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

• Improve the process by eliminating the root causes of defects.

• Control the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

• Design the process to

(1) avoid the root causes of defects and

(2) to meet customer requirements.

• Verify that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

**9. 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.

Software reliability, unlike many other quality factors, can be measured directly 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 0.999 over eight elapsed processing hours. In other words, if program X were to be executed 1000 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 999 times.

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 non - conformance 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.

**Measures of Reliability and Availability**

Early work in software reliability attempted to extrapolate the mathematics of hardware reliability theory 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 an ongoing debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet to 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 meantime-between-failure (MTBF):

**MTBF = MTTF + MTTR**

where the acronyms MTTF and MTTR are **mean-time-to-failure** and **mean-time-to-repair**, 2 respectively.

Many researchers argue that MTBF is a far more useful measure than other quality-related software metrics discussed in Chapter 23. Stated simply, an end user is concerned with failures, not with the total defect count. Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system. For example, consider a program that has been in operation for 3000 processor hours without failure. Many defects in this program may remain undetected for tens of thousand of hours before they are discovered. The MTBF of such obscure errors might be 30,000 or even 60,000 processor hours. Other defects, as yet undiscovered, might have a failure rate of 4000 or 5000 hours. Even if every one of the first category of errors (those with long MTBF) is removed, the impact on software reliability is negligible.

However, MTBF can be problematic for two reasons:

(1) it projects a time span between failures, but does not provide us with a projected failure rate, and

(2) MTBF can be misinterpreted to mean average life span even though this is not what it implies.

An alternative measure of reliability is failures-in-time (FIT)—a statistical measure of how many failures a component will have over one billion hours of operation. Therefore, 1 FIT is equivalent to one failure in every billion hours of operation.

In addition to a reliability measure, you should also 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

Availability = MTTF / MTTF + MTTR X 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.

**Software Safety**

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 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:

(1) causes uncontrolled acceleration that cannot be stopped,

(2) does not respond to depression of brake pedal (by turning off),

(3) does not engage when switch is activated, and

(4) slowly loses or gains speed. Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence. 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 and only if a set of external environmental conditions is met, the improper position of the mechanical device will cause a disastrous failure. Analysis techniques [Eri05] such as fault tree analysis, real-time logic, and Petri net models 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.

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 and its environment.

**10. Process and Project Metrics:**

Measurement enables us to gain insight into the process and the project by providing a mechanism for objective evaluation. Lord Kelvin once said:

When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science.

The software engineering community has taken Lord Kelvin’s words to heart. But not without frustration and more than a little controversy!

Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds (Chapter 23).

**What is it?**

Software process and project metrics are quantitative measures that enable you to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

**Who does it?**

Software metrics are analyzed and assessed by software managers. Measures are often collected by software engineers. Why is it important? If you don’t measure, judgment can be based only on subjective evaluation. Q UICK L OOK With measurement, trends (either good or bad) can be spotted, better estimates can be made, and true improvement can be accomplished over time.

**What are the steps?**

Begin by defining a limited set of process, project, and product measures that are easy to collect. These measures are often normalized using either size- or functionoriented metrics. The result is analyzed and compared to past averages for similar projects performed within the organization. Trends are assessed and conclusions are generated.

**What is the work product?**

A set of software metrics that provide insight into the process and understanding of the project. How do I ensure that I’ve done it right? By applying a consistent, yet simple measurement scheme that is never to be used to assess, reward, or punish individual performance.

Within the context of the software process and the projects that are conducted using the process, a software team is concerned primarily with productivity and quality metrics—measures of software development “output” as a function of effort and time applied and measures of the “fitness for use” of the work products that are produced. For planning and estimating purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help you plan and estimate more accurately?

In their guidebook on software measurement, Park, Goethert, and Florac [Par96b] note the reasons that we measure: (1) to characterize in an effort to gain an understanding “of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments”; (2) to evaluate “to determine status with respect to plans”; (3) to predict by “gaining understandings of relationships among processes and products and building models of these relationships”; and (4) to improve by “identify[ing] roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.”

Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

**11. Metrics in the Process and Project Domains**

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. Project metrics enable a software project manager to

(1) assess the status of an ongoing project,

(2) track potential risks,

(3) uncover problem areas before they go “critical,”

(4) adjust work flow or tasks, and

(5) evaluate the project team’s ability to control quality of software work products. Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement (Chapter 30). For this reason, many of the same metrics are used in both the process and project domains.

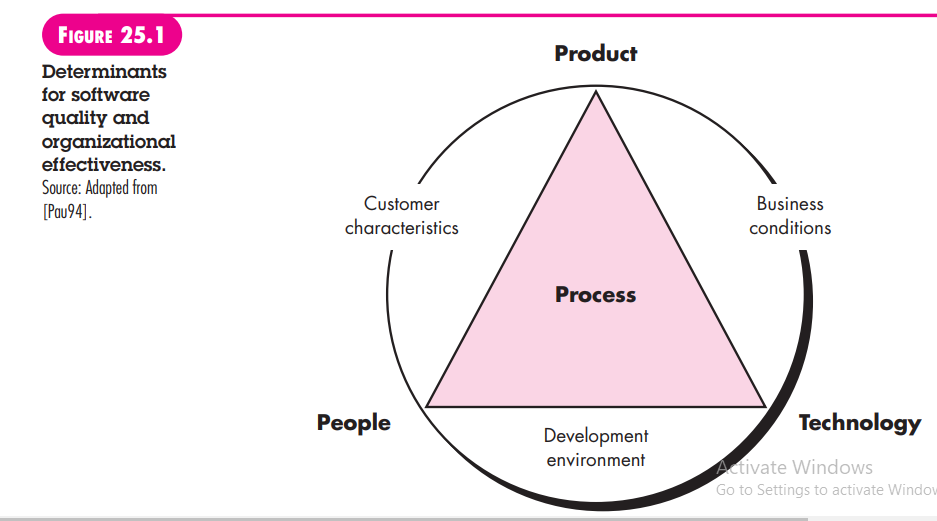
**Process Metrics and Software Process Improvement**

**Project Metrics**

**Process Metrics and Software Process Improvement**

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement (Chapter 30). But before I discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of “controllable factors in improving software quality and organizational performance” [Pau94].

Referring to Figure 25.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people has been shown [Boe81] to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.

****

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration).

You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities described in Chapter 2.

Grady [Gra92] argues that there are “private and public” uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of private metrics include defect rates (by individual), defect rates (by component), and errors found during development.

The “private process data” philosophy conforms well with the Personal Software Process approach (Chapter 2) proposed by Humphrey [Hum97]. Humphrey recognizes that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as you work to improve your software engineering approach.

Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during technical reviews, and lines of code or function points per component or function.1 The team reviews these data to uncover indicators that can improve team performance.

views these data to uncover indicators that can improve team performance. Public metrics generally assimilate information that originally was private to individuals and teams. Project-level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

Software process metrics can provide significant benefit as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady [Gra92] suggests a “software metrics etiquette” that is appropriate for both managers and practitioners as they institute a process metrics program:

• Use common sense and organizational sensitivity when interpreting metrics data.

• Provide regular feedback to the individuals and teams who collect measures and metrics.

• Don’t use metrics to appraise individuals.

• Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.

• Never use metrics to threaten individuals or teams.

• Metrics data that indicate a problem area should not be considered “negative.” These data are merely an indicator for process improvement.

• Don’t obsess on a single metric to the exclusion of other important metrics.

As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called statistical software process improvement (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects2 encountered as an application, system, or product is developed and used.

**Project Metrics**

Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project workflow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 23) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

**12. Software Measurement**

In Chapter 23, I noted that measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the “quality” of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other “–abilities” that are discussed in Chapter 14.

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

I have partitioned the software metrics domain into process, project, and product metrics and noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects?

To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

**1. Size Oriented Metrics**

**2. Function-Oriented Metrics**

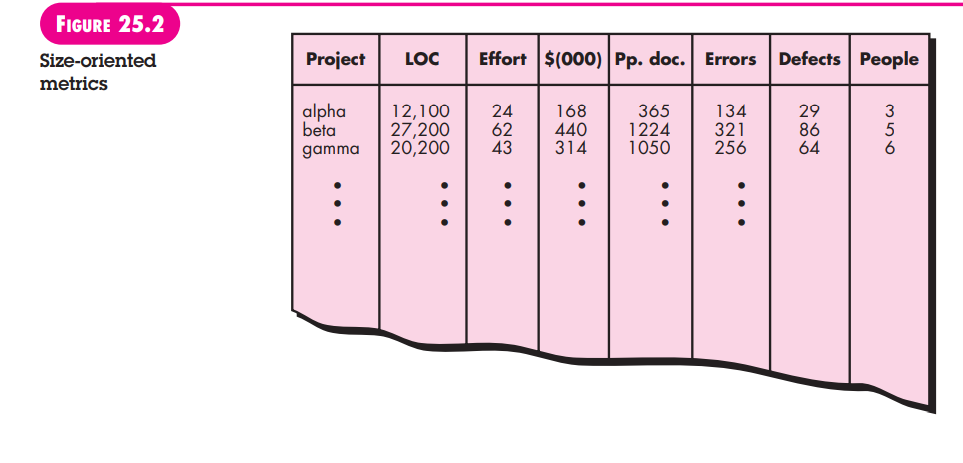
**3. Reconciling LOC and FP Metrics**

**4. Object-Oriented Metrics**

**5. Use-Case–Oriented Metrics**

**6.** **WebApps Project Metrics**

**1. Size Oriented Metrics**



Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 25.2, can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 25.2) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.

In order to develop metrics that can be assimilated with similar metrics from other projects, you can choose lines of code as a normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

• Errors per KLOC (thousand lines of code)

• Defects per KLOC

• $ per KLOC

• Pages of documentation per KLOC

In addition, other interesting metrics can be computed:

• Errors per person-month

• KLOC per person-month

• $ per page of documentation

Size-oriented metrics are not universally accepted as the best way to measure the software process. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an “artifact” of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter programs; that they cannot easily accommodate nonprocedural languages; and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).

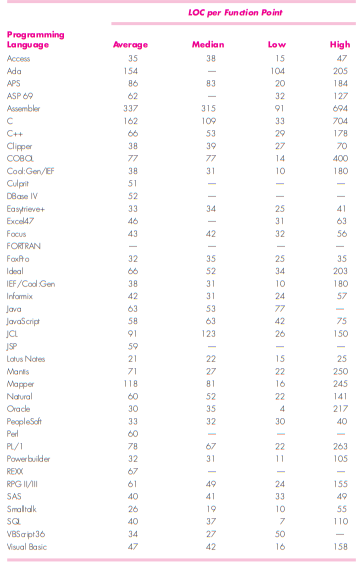
**2. Function-Oriented Metrics**

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software’s information domain and complexity. The mechanics of FP computation have been discussed in Chapter 23.3

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some “sleight of hand” in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it’s just a number.

**3. Reconciling LOC and FP Metrics**

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The following table4 [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:



A review of these data indicates that one LOC of C++ provides approximately 2.4 times the “functionality” (on average) as one LOC of C. Furthermore, one LOC of Smalltalk provides at least four times the functionality of an LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to “backfire” [Jon98] existing software to estimate the number of function points, once the total number of programming language statements are known.

LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? The answer to these questions is an emphatic “No!” The reason for this response is that many factors influence productivity, making for “apples and oranges” comparisons that can be easily misinterpreted.

Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 26), an historical baseline of information must be established.

Within the context of process and project metrics, you should be concerned primarily with productivity and quality—measures of software development “output” as a function of effort and time applied and measures of the “fitness for use” of the work products that are produced. For process improvement and project planning purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately?

**4. Object-Oriented Metrics**

Conventional software project metrics (LOC or FP) can be used to estimate objectoriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process. Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:

**Number of scenario scripts.**  
A scenario script (analogous to use cases discussed throughout Part 2 of this book) is a detailed sequence of steps that describe the interaction between the user and the application. Each script is organized into triplets of the form

**{initiator, action, participant}**

where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

**Number of key classes.**

Key classes are the “highly independent components” [Lor94] that are defined early in object-oriented analysis (Chapter 6).5 Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.  
**Number of support classes.**

Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (GUI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

**Average number of support classes per key class.**

In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

**Number of subsystems.**

A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

To be used effectively in an object-oriented software engineering environment, metrics similar to those noted above should be collected along with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the object-oriented measures and project measures will provide metrics that can aid in project estimation.

**5. Use-Case–Oriented Metrics**

Use cases6 are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP.

Like FP, the use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated. Use cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use cases can be created at vastly different levels of abstraction, there is no standard “size” for a use case. Without a standard measure of what a use case is, its application as a normalization measure (e.g., effort expended per use case) is suspect.

Researchers have suggested use-case points (UCPs) as a mechanism for estimating project effort and other characteristics. The UCP is a function of the number of actors and transactions implied by the use-case models and is analogous to the FP in some ways. If you have further interest, see [Cle06]

**6. WebApp Project Metrics**

The objective of all WebApp projects is to deliver a combination of content and functionality to the end user. Measures and metrics used for traditional software engineering projects are difficult to translate directly to WebApps. Yet, it is possible to develop a database that allows access to internal productivity and quality measures derived over a number of projects. Among the measures that can be collected are:

**Number of static Web pages.**

Web pages with static content (i.e., the end user has no control over the content displayed on the page) are the most common of all WebApp features. These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

**Number of dynamic Web pages.**

Web pages with dynamic content (i.e., end-user actions or other external factors result in customized content displayed on the page) are essential in all e-commerce applications, search engines, financial applications, and many other WebApp categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.

**Number of internal page links**.

Internal page links are pointers that provide a hyperlink to some other Web page within the WebApp. This measure provides an indication of the degree of architectural coupling within the WebApp. As the number of page links increases, the effort expended on navigational design and construction also increases.

**Number of persistent data objects.**

One or more persistent data objects (e.g., a database or data file) may be accessed by a WebApp. As the number of persistent data objects grows, the complexity of the WebApp also grows and the effort to implement it increases proportionally.

**Number of external systems interfaced.**

WebApps must often interface with “backroom” business applications. As the requirement for interfacing grows, system complexity and development effort also increase.

**Number of static content objects.**

Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a single Web page.

**Number of dynamic content objects.**

Dynamic content objects are generated based on end-user actions and encompass internally generated text based, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a single Web page.

**Number of executable functions.**

An executable function (e.g., a script or applet) provides some computational service to the end user. As the number of executable functions increases, modeling and construction effort also increase.

Each of the preceding measures can be determined at a relatively early stage. For example, you can define a metric that reflects the degree of end-user customization that is required for the WebApp and correlate it to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted. To accomplish this, you define

Nsp = number of Static Web pages

Ndp = number of Dynamic Web pages

Then,

Customization index C = Ndp/Ndp+ Nsp

The value of C ranges from 0 to 1. As C grows larger, the level of WebApp customization becomes a significant technical issue.

Similar WebApp metrics can be computed and correlated with project measures such as effort expended, errors and defects uncovered, and models or documentation pages produced. As the database grows (after a number of projects have been completed), relationships between the WebApp measures and project measures will provide indicators that can aid in project estimation.

13. Metrics for Software Quality

The overriding goal of software engineering is to produce a high-quality system, application, or product within a time frame that satisfies a market need. To achieve this goal, you must apply effective methods coupled with modern tools within the context of a mature software process. In addition, a good software engineer (and good software engineering managers) must measure if high quality is to be realized.

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. You can use measurement to assess the quality of the requirements and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time assessment, you apply product metrics (Chapter 23) to evaluate the quality of software engineering work products in objective, rather than subjective ways.

A project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are combined to provide projectlevel results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities.

Metrics such as work product errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the defect removal efficiency (DRE) for each process framework activity. DRE is discussed in Section 25.3.3.

**1. Measuring Quality**

**2. Defect Removal Efficiency**

**1. Measuring Quality**

Although there are many measures of software quality,8 correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb [Gil88] suggests definitions and measures for each.

**Correctness.**

A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

**Maintainability.**

Software maintenance and support accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. There is no way to measure maintainability directly; therefore, you must use indirect measures. A simple time-oriented metric is mean-time-to-change (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users. On average, programs that are maintainable will have a lower MTTC (for equivalent types of changes) than programs that are not maintainable.

**Integrity.**

Software integrity has become increasingly important in the age of cyber terrorists and hackers. This attribute measures a system’s ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documentation.

To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as:

**Integrity [1 (threat X(1 - security))]**

For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high). If, on the other hand, the threat probability is 0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

**Usability**.

If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics presented in Chapter 11.

The four factors just described are only a sampling of those that have been proposed as measures for software quality. Chapter 23 considers this topic in additional detail.

**2. Defect Removal Efficiency**

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

DRE = E / E + D

where E is the number of errors found before delivery of the software to the end user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1 as E increases for a given value of D. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

DRE can also be used within the project to assess a team’s ability to find errors before they are passed to the next framework activity or software engineering action. For example, requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found). When used in this context, we redefine DRE as

DREi = Ei / Ei + Ei + 1

where Ei is the number of errors found during software engineering action i and Ei+ 1 is the number of errors found during software engineering action i + 1 that are traceable to errors that were not discovered in software engineering action i.

A quality objective for a software team (or an individual software engineer) is to achieve DREi that approaches 1. That is, errors should be filtered out before they are passed on to the next activity or action.

14. Establishing a Software Metrics Program.

The Software Engineering Institute has developed a comprehensive guidebook [Par96b] for establishing a “goal-driven” software metrics program. The guidebook suggests the following steps:

1. Identify your business goals.

2. Identify what you want to know or learn.

3. Identify your sub-goals.

4. Identify the entities and attributes related to your sub-goals.

5. Formalize your measurement goals.

6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.

7. Identify the data elements that you will collect to construct the indicators that help answer your questions.

8. Define the measures to be used, and make these definitions operational.

9. Identify the actions that you will take to implement the measures.

10. Prepare a plan for implementing the measures.

A detailed discussion of these steps is best left to the SEI’s guidebook. However, a brief overview of key points is worthwhile.

Because software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider the SafeHome product. Working as a team, software engineering and business managers develop a list of prioritized business goals:

1. Improve our customers’ satisfaction with our products.

2. Make our products easier to use.

3. Reduce the time it takes us to get a new product to market.

4. Make support for our products easier.

5. Improve our overall profitability

The software organization examines each business goal and asks: “What activities do we manage, execute, or support and what do we want to improve within these activities?” To answer these questions the SEI recommends the creation of an “entity-question list” in which all things (entities) within the software process that are managed or influenced by the software organization are noted.

Examples ofentities include development resources, work products, source code, test cases, change requests, software engineering tasks, and schedules. For each entity listed, software people develop a set of questions that assess quantitative characteristics of the entity (e.g., size, cost, time to develop). The questions derived as a consequence of the creation of an entity-question list lead to the derivation of a set of subgoals that relate directly to the entities created and the activities performed as part of the software process

Consider the fourth goal: “Make support for our products easier.” The following list of questions might be derived for this goal [Par96b]:

• Do customer change requests contain the information we require to adequately evaluate the change and then implement it in a timely manner?

• How large is the change request backlog?

• Is our response time for fixing bugs acceptable based on customer need?

• Is our change control process (Chapter 22) followed?

• Are high-priority changes implemented in a timely manner?

Based on these questions, the software organization can derive the following sub-goal: Improve the performance of the change management process. The software process entities and attributes that are relevant to the sub-goal are identified, and the measurement goals associated with them are delineated.

The SEI [Par96b] provides detailed guidance for steps 6 through 10 of its goal-driven measurement approach. In essence, you refine measurement goals into questions that are further refined into entities and attributes that are then refined into metrics.