CHAPTER

16 SOFTWARE QUALITY ASSURANCE

KEY CONCEPTS

The software engineering approach described in this book works toward a single goal: to produce on-time, high-quality software. Yet many readers will be challenged by the question: "What is software quality?" Philip Crosby [Cro79], in his landmark book on quality, provides a wry answer to this question:

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

QUICK LOOK

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

Who does it? Everyone involved in the software engineering process is responsible for quality.

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

What are the steps? Before software quality assurance (SQA) activities can be initiated, it is important to define *software quality* at a number of different levels of abstraction. Once you understand what quality is, a software team must identify a set of SQA activities that will filter errors out of work products before they are passed on.

What is the work product? A Software Quality Assurance Plan is created to define a software team's SQA strategy. During modeling and coding, the primary SQA work product is the output of technical reviews (Chapter 15). During testing (Chapters 17 through 20), test plans and procedures are produced. Other work products associated with process improvement may also be generated.

How do I ensure that I've done it right? Find errors before they become defects! That is, work to improve your defect removal efficiency (Chapter 23), thereby reducing the amount of rework that your software team has to perform.

Indeed, quality is a challenging concept—one that I addressed in some detail in Chapter 14.[1]

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 multitiered 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), and (6) measurement and reporting mechanisms.

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

16.1 Background Issues

Quality control and assurance are essential activities for any business that produces products to be used by others. Prior to the twentieth century, quality control was the sole responsibility of the craftsperson who built a product. As time passed and mass production techniques became commonplace, quality control became an activity performed by people other than the ones who built the product.

Quote:
u made too
ny wrong
takes."
i Berra

The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world. During the 1940s, more formal approaches to quality control were suggested. These relied on measurement and continuous process improvement [Dem86] as key elements of quality management.

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

The history of quality assurance in software development parallels the history of quality in hardware manufacturing. During the early days of computing (1950s and 1960s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world [IEE93]. Extending the definition presented earlier, software quality assurance is a "planned and systematic pattern of actions" [Sch98c] that are required to ensure high quality in software. The scope of quality assurance responsibility might best be characterized by paraphrasing a once-popular automobile commercial: "Quality Is Job #1." The implication for software is that many different constituencies

[1] If you have not read Chapter 14, you should do so now.

have software quality assurance responsibility—software engineers, project managers, customers, salespeople, and the individuals who serve within an SQA group. The SQA group serves as the customer's in-house representative. That is, the people who perform SQA must look at the software from the customer's point of view. Does the software adequately meet the quality factors noted in Chapter 14? Has software development been conducted according to preestablished standards? Have technical disciplines properly performed their roles as part of the SQA activity? The SQA group attempts to answer these and other questions to ensure that software quality is maintained.

16.2 Elements of Software Quality Assurance

WebRef

An in-depth discussion of SQA, including a wide array of limitations, can be obtained at www.swqual.com/newsletter/vol1/no1/no1.html.

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]:

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.

Note:
"Excellence is the unlimited ability to improve the quality of what you have to offer."
Rick Pitino

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 human-rated 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.

INFO

Quality Management Resources

There are dozens of quality management resources available on the Web, including professional societies, standards organizations, and general information sources. The sites that follow provide a good starting point:

American Society for Quality (ASQ) Software Division
www.asq.org/software

Association for Computer Machinery **www.acm.org**

Data and Analysis Center for Software (DACS)
www.dacs.dtic.mil/

International Organization for Standardization (ISO)
www.iso.ch

ISO SPICE
www.isospice.com

Malcolm Baldridge National Quality Award
www.quality.nist.gov
Software Engineering Institute
www.sei.cmu.edu/
Software Testing and Quality Engineering
www.stickyminds.com
Six Sigma Resources
www.isixsigma.com/
www.asq.org/sixsigma/

TickIT International: Quality certification topics
www.tickit.org/international.htm
Total Quality Management (TQM)
General information:
www.gslis.utexas.edu/~rpollock/tqm.html
Articles: **www.work911.com/tqmarticles.htm**
Glossary:
www.quality.org/TQM-MSI/TQM-glossary.html

16.3 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.

16.3.1 SQA Tasks

The charter of the SQA group is to assist the software team in achieving a high-quality 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:

? What is the role of an SQA group?

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.

16.3.2 Goals, Attributes, and Metrics

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

Quote:

"Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives."

William A. Foster

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.

FIGURE 16.1 Software quality goals, attributes, and metrics
Source: Adapted from [Hyo96].

Goal	Attribute	Metric
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
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	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	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

16.4 FORMAL APPROACHES TO SQA

In the preceding sections, I have argued that software quality is everyone's job and that it can be achieved through competent software engineering practice as well as through the application of technical reviews, a multi-tiered testing strategy, better control of software work products and the changes made to them, and the application of accepted software engineering standards. In addition, quality can be defined

in terms of a broad array of quality attributes and measured (indirectly) using a variety of indices and metrics.

Over the past three decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements (Chapter 21) is available. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

Attempts to prove programs correct are not new. Dijkstra [Dij76a] and Linger, Mills, and Witt [Lin79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 10).

16.5 Statistical Software Quality Assurance

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

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., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
4. Once the vital few causes have been identified, move to correct the problems that have caused the 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.

16.5.1 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)

Quote:

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

Lowell Arthur

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

FIGURE 16.2
Data collection for statistical SQA

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

have achieved a 50 percent reduction per year in defects after applying these techniques.

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!*

16.5.2 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:

? What are the core steps of the Six Sigma methodology?

- *Define* customer requirements and deliverables and project goals via well-defined 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.

DMAIC

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.

DMADV

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

A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject. If you have further interest, see [ISI08], [Pyz03], and [Sne03].

16.6 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.

Note:

"The unavoidable price of reliability is simplicity."

C. A. R. Hoare

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 nonconformance to software requirements. Yet, even within this definition, there are gradations. Failures can be only annoying or catastrophic. One failure can be corrected within seconds, while another requires weeks or even months to correct. Complicating the issue even further, the correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

16.6.1 Measures of Reliability and Availability

KEY POINT

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

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.

KEY POINT

It is important to note that MTBF and related measures are based on CPU time, not wall clock time.

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

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

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

2 Although debugging (and related corrections) may be required as a consequence of failure, in many cases the software will work properly after a restart with no other change.

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.

ADVICE

Few aspects of reliability (not discussed here) have anything to do with time. For example, predicting downtime (or support functions) causes the software to be unavailable.

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

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\%$$

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

16.6.2 Software Safety

Quote:

"The safety of the people shall be the highest law."

Cicero

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.[3] To be effective, software must be analyzed in the context of the entire

3 This approach is similar to the risk analysis methods described in Chapter 28. The primary difference is the emphasis on technology issues rather than project-related topics.

Quote:

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

E. J. Smith, Captain of the Titanic

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.

WebRef

A worthwhile collection of papers on software safety can be found at www.safeware-eng.com/.

A comprehensive discussion of software safety is beyond the scope of this book. If you have further interest in software safety and related system issues, see [Smi05], [Dun02], and [Lev95].

16.7 The ISO 9000 Quality Standards[4]

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

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

The requirements delineated by ISO 9001:2000 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques. In order for a software organization to become registered to ISO 9001:2000, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed. If you desire further information on ISO 9001:2000, see [Ant06], [Mut03], or [Dob04].

The ISO 9001:2000 Standard

INFO

The following outline defines the basic elements of the ISO 9001:2000 standard. Comprehensive information on the standard can be obtained from the International Organization for Standardization (**www.iso.ch**) and other Internet sources (e.g., **www.praxiom.com**).

Establish the elements of a quality management system.
- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system.
- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.
- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction.
- Define a quality plan that addresses objectives, responsibilities, and authority.
- Define communication mechanisms among stakeholders.

Establish review mechanisms for the quality management system.
- Identify review methods and feedback mechanisms.
- Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

Establish control mechanisms.
- For planning
- For customer requirements
- For technical activities (e.g., analysis, design, testing)
- For project monitoring and management

Define methods for remediation.
- Assess quality data and metrics.
- Define approach for continuous process and quality improvement.

16.8 The SQA Plan

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or by the software team if an SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

A standard for SQA plans has been published by the IEEE [IEE93]. The standard recommends a structure that identifies: (1) the purpose and scope of the plan, (2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA, (3) all applicable standards and practices that are applied during the software process, (4) SQA actions and tasks

(including reviews and audits) and their placement throughout the software process, (5) the tools and methods that support SQA actions and tasks, (6) software configuration management (Chapter 22) procedures, (7) methods for assembling, safeguarding, and maintaining all SQA-related records, and (8) organizational roles and responsibilities relative to product quality.

SOFTWARE TOOLS

Software Quality Management

Objective: The objective of SQA tools is to assist a project team in assessing and improving the quality of software work product.

Mechanics: Tools mechanics vary. In general, the intent is to assess the quality of a specific work product. Note: A wide array of software testing tools (see Chapters 17 through 20) are often included within the SQA tools category.

Representative Tools:[5]

ARM, developed by NASA (**satc.gsfc.nasa.gov/tools/index.html**), provides measures that can be used to assess the quality of a software requirements document.

QPR ProcessGuide and Scorecard, developed by QPR Software (**www.qpronline.com**), provides support for Six Sigma and other quality management approaches.

Quality Tools and Templates, developed by iSixSigma (**www.isixsigma.com/tt/**), describes a wide array of useful tools and methods for quality management.

NASA Quality Resources, developed by the Goddard Space Flight Center (**sw-assurance.gsfc.nasa.gov/index.php**) provides useful forms, templates, checklists, and tools for SQA.

16.9 SUMMARY

Software quality assurance is a software engineering umbrella activity that is applied at each step in the software process. SQA encompasses procedures for the effective application of methods and tools, oversight of quality control activities such as technical reviews and software testing, procedures for change management, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

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

In summary, you should note the words of Dunn and Ullman [Dun82]: "Software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable managerial and technological space of software engineering." The ability to ensure quality is the measure of a mature engineering discipline. When the mapping is successfully accomplished, mature software engineering is the result.

---

5 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

CHAPTER

30 SOFTWARE PROCESS IMPROVEMENT

KEY CONCEPTS

Long before the phrase "software process improvement" was widely used, I worked with major corporations in an attempt to improve the state of their software engineering practices. As a consequence of my experiences, I wrote a book entitled *Making Software Engineering Happen* [Pre88]. In the preface of that book I made the following comment:

> For the past ten years I have had the opportunity to help a number of large companies implement software engineering practices. The job is difficult and rarely goes as smoothly as one might like—but when it succeeds, the results are profound. Software projects are more likely to be completed on time. Communication between all constituencies involved in software development is improved. The level of confusion and chaos that is often prevalent for large software projects is reduced substantially. The number of errors encountered by the customer drops substantially. The credibility of the software organization increases. And management has one less problem to worry about.
>
> But all is not sweetness and light. Many companies attempt to implement software engineering practice and give up in frustration. Others do it half-way and never see the benefits noted above. Still others do it in a heavy-handed fashion that results in open rebellion among technical staff and managers and subsequent loss of morale.

QUICK LOOK

What is it? Software process improvement encompasses a set of activities that will lead to a better software process and, as a consequence, higher-quality software delivered in a more timely manner.

Who does it? The people who champion SPI come from three groups: technical managers, software engineers, and individuals who have quality assurance responsibility.

Why is it important? Some software organizations have little more than an ad hoc software process. As they work to improve their software engineering practices, they must address weaknesses in their existing process and try to improve their approach to software work.

What are the steps? The approach to SPI is iterative and continuous, but it can be viewed in five steps: (1) assessment of the current software process, (2) education and training of practitioners and managers, (3) selection and justification of process elements, software engineering methods, and tools, (4) implementation of the SPI plan, and (5) evaluation and tuning based on the results of the plan.

What is the work product? Although there are many intermediate SPI work products, the end result is an improved software process that leads to higher-quality software.

How do I ensure that I've done it right? The software your organization produces will be delivered with fewer defects, rework at each stage of the software process will be reduced, and on-time delivery will become much more likely.

Although those words were written more than 20 years ago, they remain equally true today.

As we move into the second decade of the twenty-first century, most major software engineering organizations have attempted to "make software engineering happen." Some have implemented individual practices that have helped to improve the quality of the product they build and the timeliness of their delivery. Others have established a "mature" software process that guides their technical and project management activities. But others continue to struggle. Their practices are hit-and-miss, and their process is ad hoc. Occasionally, their work is spectacular, but in the main, each project is an adventure, and no one knows whether it will end badly or well.

So, which of these two cohorts needs software process improvement? The answer (which may surprise you) is *both*. Those that have succeeded in making software engineering happen cannot become complacent. They must work continually to improve their approach to software engineering. And those that struggle must begin their journey down the road toward improvement.

30.1 What Is SPI?

KEY POINT

SPI implies a defined software process, an organizational approach, and a strategy for improvement.

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Because SPI is not free, it must deliver a return on investment. The effort and time that is required to implement an SPI strategy must pay for itself in some measurable way. To do this, the results of improved process and practice must lead to a reduction in software "problems" that cost time and money. It must reduce the number of defects that are delivered to end users, reduce the amount of rework due to quality problems, reduce the costs associated with software maintenance and support (Chapter 29), and reduce the indirect costs that occur when software is delivered late.

Quote:

"Much of the software crisis is self-inflicted, as when a CIO says, 'I'd rather have it wrong than have it late. We can always fix it later.'"

Mark Paulk

30.1.1 Approaches to SPI

Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of SPI frameworks. An *SPI framework* defines (1) a set of characteristics that must be present if an effective software process is to be achieved, (2) a method for assessing whether those characteristics are present, (3) a mechanism for summarizing the results of any assessment, and (4) a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

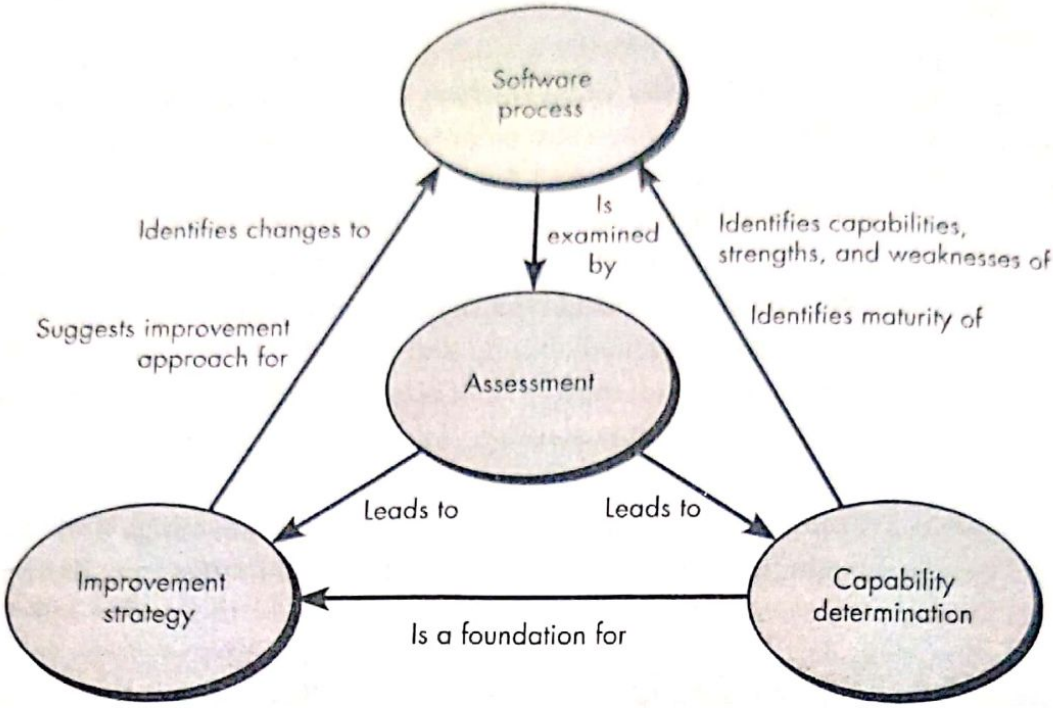

FIGURE 30.1 Elements of an SPI framework. *Source: Adapted from [Rou02].*

An SPI framework assesses the "maturity" of an organization's software process and provides a qualitative indication of a maturity level. In fact, the term "maturity model" (Section 30.1.2) is often applied. In essence, the SPI framework encompasses a maturity model that in turn incorporates a set of process quality indicators that provide an overall measure of the process quality that will lead to product quality.

Figure 30.1 provides an overview of a typical SPI framework. The key elements of the framework and their relationship to one another are shown.

? What groups champion an PI effort?

You should note that there is no universal SPI framework. In fact, the SPI framework that is chosen by an organization reflects the constituency that is championing the SPI effort. Conradi [Con96] defines six different SPI support constituencies:

Quality certifiers. Process improvement efforts championed by this group focus on the following relationship:

Quality(*Process*) ⇒ **Quality**(*Product*)

Their approach is to emphasize assessment methods and to examine a well-defined set of characteristics that allow them to determine whether the process exhibits quality. They are most likely to adopt a process framework such as the CMM, SPICE, TickIT, or Bootstrap.[1]

Formalists. This group wants to understand (and when possible, optimize) process workflow. To accomplish this, they use process modeling languages

1 Each of these SPI frameworks is discussed later in this chapter.

(PMLs) to create a model of the existing process and then design extensions or modifications that will make the process more effective.

Tool advocates. This group insists on a tool-assisted approach to SPI that models workflow and other process characteristics in a manner that can be analyzed for improvement.

Practitioners. This constituency uses a pragmatic approach, "emphasizing mainstream project-, quality- and product management, applying project-level planning and metrics, but with little formal process modeling or enactment support" [Con96].

Reformers. The goal of this group is organizational change that might lead to a better software process. They tend to focus more on human issues (Section 30.5) and emphasize measures of human capability and structure.

Ideologists. This group focuses on the suitability of a particular process model for a specific application domain or organizational structure. Rather than typical software process models (e.g., iterative models), ideologists would have a greater interest in a process that would, say, support reuse or reengineering.

? "What are the different constituencies that support SPI?"

As an SPI framework is applied, the sponsoring constituency (regardless of its overall focus) must establish mechanisms to: (1) support technology transition, (2) determine the degree to which an organization is ready to absorb process changes that are proposed, and (3) measure the degree to which changes have been adopted.

30.1.2 Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization. That is, an indication of the quality of the software process, the degree to which practitioner's understand and apply the process, and the general state of software engineering practice. This is accomplished using some type of ordinal scale.

For example, the Software Engineering Institute's *Capability Maturity Model* (Section 30.4) suggests five levels of maturity [Sch96]:

Level 5, Optimized—The organization has quantitative feedback systems in place to identify process weaknesses and strengthen them pro-actively. Project teams analyze defects to determine their causes; software processes are evaluated and updated to prevent known types of defects from recurring.

Level 4, Managed—Detailed software process and product quality metrics establish the quantitative evaluation foundation. Meaningful variations in process performance can be distinguished from random noise, and trends in process and product qualities can be predicted.

Level 3, Defined—Processes for management and engineering are documented, standardized, and integrated into a standard software process for the organization.

KEY POINT
A maturity model defines levels of software process competence and implementation.

All projects use an approved, tailored version of the organization's standard software process for developing software.

Level 2, Repeatable—Basic project management processes are established to track cost, schedule, and functionality. Planning and managing new products is based on experience with similar projects.

Level 1, Initial—Few processes are defined, and success depends more on individual heroic efforts than on following a process and using a synergistic team effort.

The CMM maturity scale goes no further, but experience indicates that many organizations exhibit levels of "process immaturity" [Sch96] that undermine any rational attempt at improving software engineering practices. Schorsch [Sch06] suggests four levels of immaturity that are often encountered in the real world of software development organizations:

? "How do you recognize an organization that will resist SPI efforts?"

Level 0, Negligent—Failure to allow successful development process to succeed. All problems are perceived to be technical problems. Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process. Reliance on silver pellets.

Level -1, Obstructive—Counterproductive processes are imposed. Processes are rigidly defined and adherence to the form is stressed. Ritualistic ceremonies abound. Collective management precludes assigning responsibility. Status quo über alles.

Level -2, Contemptuous—Disregard for good software engineering institutionalized. Complete schism between software development activities and software process improvement activities. Complete lack of a training program.

Level -3, Undermining—Total neglect of own charter, conscious discrediting of peer organization's software process improvement efforts. Rewarding failure and poor performance.

Schorsch's immaturity levels are toxic for any software organization. If you encounter any one of them, attempts at SPI are doomed to failure.

The overriding question is whether maturity scales, such as the one proposed as part of the CMM, provide any real benefit. I think that they do. A maturity scale provides an easily understood snapshot of process quality that can be used by practitioners and managers as a benchmark from which improvement strategies can be planned.

30.1.3 Is SPI for Everyone?

For many years, SPI was viewed as a "corporate" activity—a euphemism for something that only large companies perform. But today, a significant percentage of all software development is being performed by companies that employ fewer than 100 people. Can a small company initiate SPI activities and do it successfully?

ADVICE

a specific process model or SPI approach feels like overkill for your organization, it probably is.

There are substantial cultural differences between large software development organizations and small ones. It should come as no surprise that small organizations are more informal, apply fewer standard practices, and tend to be self-organizing. They also tend to pride themselves on the "creativity" of individual members of the

software organization, and initially view an SPI framework as overly bureaucratic and ponderous. Yet, process improvement is as important for a small organization as it is for a large one.

Within small organizations the implementation of an SPI framework requires resources that may be in short supply. Managers must allocate people and money to make software engineering happen. Therefore, regardless of the size of the software organization, it's reasonable to consider the business motivation for SPI.

SPI will be approved and implemented only after its proponents demonstrate financial leverage [Bir98]. Financial leverage is demonstrated by examining technical benefits (e.g., fewer defects delivered to the field, reduced rework, lower maintenance costs, or more rapid time-to-market) and translating them into dollars. In essence, you must show a realistic return on investment (Section 30.7) for SPI costs.

30.2 The SPI Process

The hard part of SPI isn't the definition of characteristics that define a high-quality software process or the creation of a process maturity model. Those things are relatively easy. Rather, the hard part is establishing a consensus for initiating SPI and defining an ongoing strategy for implementing it across a software organization.

The Software Engineering Institute has developed IDEAL—"an organizational improvement model that serves as a roadmap for initiating, planning, and implementing improvement actions" [SEI08]. IDEAL is representative of many process models for SPI, defining five distinct activities—initiating, diagnosing, establishing, acting, and learning—that guide an organization through SPI activities.

In this book, I present a somewhat different road map for SPI, based on the process model for SPI originally proposed in [Pre88]. It applies a commonsense philosophy that requires an organization to (1) look in the mirror, (2) then get smarter so it can make intelligent choices, (3) select the process model (and related technology elements) that best meets its needs, (4) instantiate the model into its operating environment and its culture, and (5) evaluate what has been done. These five activities (discussed in the subsections[2] that follow) are applied in an iterative (cyclical) manner in an effort to foster continuous process improvement.

30.2.1 Assessment and Gap Analysis

Any attempt to improve your current software process without first assessing the efficacy of current framework activities and associated software engineering practices would be like starting on a long journey to a new location with no idea where you are starting from. You'd depart with great flourish, wander around trying to get your bearings, expend lots of energy and endure large doses of frustration, and likely,

2 Some of the content in these sections has been adapted from [Pre88] with permission.

decide you really didn't want to travel anyway. Stated simply, before you begin any journey, it's a good idea to know precisely where you are.

ADVICE

Be sure to understand your strengths as well as your weaknesses. If you're smart, you'll build on the strengths.

The first road-map activity, called *assessment*, allows you to get your bearings. The intent of assessment is to uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process.

Assessment examines a wide range of actions and tasks that will lead to a high-quality process. For example, regardless of the process model that is chosen, the software organization must establish generic mechanisms such as: defined approaches for customer communication; established methods for representing user requirements; defining a project management framework that includes scoping, estimation, scheduling, and project tracking; risk analysis methods; change management procedures; quality assurance and control activities including reviews; and many others. Each is considered within the context of the framework and umbrella activities (Chapter 2) that have been established and is assessed to determine whether each of the following questions has been addressed:

- Is the objective of the action clearly defined?
- Are work products required as input and produced as output identified and described?
- Are the work tasks to be performed clearly described?
- Are the people who must perform the action identified by role?
- Have entry and exit criteria been established?
- Have metrics for the action been established?
- Are tools available to support the action?
- Is there an explicit training program that addresses the action?
- Is the action performed uniformly for all projects?

Although the questions noted imply a *yes* or *no* answer, the role of assessment is to look behind the answer to determine whether the action in question is being performed in a manner that would conform to best practice.

As the process assessment is conducted, you (or those who have been hired to perform the assessment) should also focus on the following attributes:

? "What generic attributes do you look for during assessment?"

Consistency. Are important activities, actions, and tasks applied consistently across all software projects and by all software teams?

Sophistication. Are management and technical actions performed with a level of sophistication that implies a thorough understanding of best practice?

Acceptance. Is the software process and software engineering practice widely accepted by management and technical staff?

Commitment. Has management committed the resources required to achieve consistency, sophistication, and acceptance?

The difference between local application and best practice represents a "gap" that offers opportunities for improvement. The degree to which consistency, sophistication, acceptance, and commitment are achieved indicates the amount of cultural change that will be required to achieve meaningful improvement.

30.2.2 Education and Training

Although few software people question the benefit of an agile, organized software process or solid software engineering practices, many practitioners and managers do not know enough about either subject.[3] As a consequence, inaccurate perceptions of process and practice lead to inappropriate decisions when an SPI framework is introduced. It follows that a key element of any SPI strategy is education and training for practitioners, technical managers and more senior managers who have direct contact with the software organization. Three types of education and training should be conducted:

Try to provide "just-in-time" training targeted to the real needs of a software team.

Generic concepts and methods. Directed toward both managers and practitioners, this category stresses both process and practice. The intent is to provide professionals with the intellectual tools they need to apply the software process effectively and to make rational decisions about improvements to the process.

Specific technology and tools. Directed primarily toward practitioners, this category stresses technologies and tools that have been adopted for local use. For example, if UML has been chosen for analysis and design modeling, a training curriculum for software engineering using UML would be established.

Business communication and quality-related topics. Directed toward all stakeholders, this category focuses on "soft" topics that help enable better communication among stakeholders and foster a greater quality focus.

In a modern context, education and training can be delivered in a variety of different ways. Everything from podcasts, to Internet-based training (e.g., [QAI08]), to DVDs, to classroom courses can be offered as part of an SPI strategy.

As you make your choices, be sure to consider the culture of your organization and the level of acceptance that each choice will likely elicit.

30.2.3 Selection and Justification

Once the initial assessment activity[4] has been completed and education has begun, a software organization should begin to make choices. These choices occur during a *selection and justification activity* in which process characteristics and specific software engineering methods and tools are chosen to populate the software process.

First, you should choose the process model (Chapters 2 and 3) that best fits your organization, its stakeholders, and the software that you build. You should decide

3 If you've spent time reading this book, you won't be one of them!

4 In actuality, assessment is an ongoing activity. It is conducted periodically in an effort to determine whether the SPI strategy has achieved its immediate goals and to set the stage for future improvement.

which of the set of framework activities will be applied, the major work products that will be produced, and the quality assurance checkpoints that will enable your team to assess progress. If the SPI assessment activity indicates specific weaknesses (e.g., no formal SQA functions), you should focus attention on process characteristics that will address these weaknesses directly.

Next, develop a work breakdown for each framework activity (e.g., modeling), defining the task set that would be applied for a typical project. You should also consider the software engineering methods that can be applied to achieve these tasks. As choices are made, education and training should be coordinated to ensure that understanding is reinforced.

Ideally, everyone works together to select various process and technology elements and moves smoothly toward the installation or migration activity (Section 30.2.4). In reality, selection can be a rocky road. It is often difficult to achieve consensus among different constituencies. If the criteria for selection are established by committee, people may argue endlessly about whether the criteria are appropriate and whether a choice truly meets the criteria that have been established.

It is true that a bad choice can do more harm than good, but "paralysis by analysis" means that little if any progress occurs and process problems remain. As long as the process characteristic or technology element has a good chance at meeting an organization's needs, it's sometimes better to pull the trigger and make a choice, rather than waiting for the optimal solution.

Once a choice is made, time and money must be expended to instantiate it within an organization, and these resource expenditures should be justified. A discussion of cost justification and return on investment for SPI is presented in Section 30.7.

30.2.4 Installation/Migration

Installation is the first point at which a software organization feels the effects of changes implemented as a consequence of the SPI road map. In some cases, an entirely new process is recommended for an organization. Framework activities, software engineering actions, and individual work tasks must be defined and installed as part of a new software engineering culture. Such changes represent a substantial organizational and technological transition and must be managed very carefully.

In other cases, changes associated with SPI are relatively minor, representing small, but meaningful modifications to an existing process model. Such changes are often referred to as *process migration*. Today, many software organizations have a "process" in place. The problem is that it doesn't work in an effective manner. Therefore, an incremental *migration* from one process (that doesn't work as well as desired) to another process is a more effective strategy.

Installation and migration are actually *software process redesign* (SPR) activities. Scacchi [Sca00] states that "SPR is concerned with identification, application, and

refinement of new ways to dramatically improve and transform software processes." When a formal approach to SPR is initiated, three different process models are considered: (1) the existing ("as is") process, (2) a transitional ("here to there") process, and (3) the target ("to be") process. If the target process is significantly different from the existing process, the only rational approach to installation is an incremental strategy in which the transitional process is implemented in steps. The transitional process provides a series of way-points that enable the software organization's culture to adapt to small changes over a period of time.

30.2.5 Evaluation

Although it is listed as the last activity in the SPI road map, *evaluation* occurs throughout SPI. The evaluation activity assesses the degree to which changes have been instantiated and adopted, the degree to which such changes result in better software quality or other tangible process benefits, and the overall status of the process and the organizational culture as SPI activities proceed.

Both qualitative factors and quantitative metrics are considered during the evaluation activity. From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes. Quantitative metrics (Chapter 25) are collected from projects that have used the transitional or "to be" process and compared with similar metrics that were collected for projects that were conducted under the "as is" process.

30.2.6 Risk Management for SPI

KEY POINT
SPI often fails because risks were not properly considered and no contingency planning occurred.

SPI is a risky undertaking. In fact, more than half of all SPI efforts end in failure. The reasons for failure vary greatly and are organizationally specific. Among the most common risks are: a lack of management support, cultural resistance by technical staff, a poorly planned SPI strategy, an overly formal approach to SPI, selection of an inappropriate process, a lack of buy-in by key stakeholders, an inadequate budget, a lack of staff training, organizational instability, and a myriad of other factors. The role of those chartered with the responsibility for SPI is to analyze likely risks and develop an internal strategy for mitigating them.

A software organization should manage risk at three key points in the SPI process [Sta97b]: prior to the initiation of the SPI road map, during the execution of SPI activities (assessment, education, selection, installation), and during the evaluation activity that follows the instantiation of some process characteristic. In general, the following categories [Sta97b] can be identified for SPI risk factors: budget and cost, content and deliverables, culture, maintenance of SPI deliverables, mission and goals, organizational management, organizational stability, process stakeholders, schedule for SPI development, SPI development environment, SPI development process, SPI project management, and SPI staff.

Within each category, a number of generic risk factors can be identified. For example, the organizational culture has a strong bearing on risk. The following generic risk factors[5] can be defined for the culture category [Sta97b]:

- Attitude toward change, based on prior efforts to change
- Experience with quality programs, level of success
- Action orientation for solving problems versus political struggles
- Use of facts to manage the organization and business
- Patience with change; ability to spend time socializing
- Tools orientation—expectation that tools can solve the problems
- Level of "planfulness"—ability of organization to plan
- Ability of organization members to participate with various levels of organization openly at meetings
- Ability of organization members to manage meetings effectively
- Level of experience in organization with defined processes

Using the risk factors and generic attributes as a guide, risk exposure is computed in the following manner:

$$\text{Exposure} = (\text{risk probability}) \times (\text{estimated loss})$$

A risk table (Chapter 28) can be developed to isolate those risks that warrant further management attention.

30.2.7 Critical Success Factors

In Section 30.2.6, I noted that SPI is a risky endeavor and that the failure rate for companies that try to improve their process is distressingly high. Organizational risks, people risks, and project management risks present challenges for those who lead any SPI effort. Although risk management is important, it's equally important to recognize those critical factors that lead to success.

After examining 56 software organizations and their SPI efforts, Stelzer and Mellis [Ste99] identify a set of critical success factors (CSFs) that must be present if SPI is to succeed. The top five CSFs are presented in this section.

What critical success rs are crucial ccessful

Management commitment and support. Like most activities that precipitate organizational and cultural change, SPI will succeed only if management is actively involved. Senior business managers should recognize the importance of software to their company and be active sponsors of the SPI effort. Technical managers should be heavily involved in the development of the local SPI strategy. As the authors of the study note: "Software process improvement is not feasible without investing time, money, and effort" [Ste99]. Management commitment and support are essential to sustain that investment.

5 Risk factors for each of the risk categories noted in this section can be found in [Sta97b].

Staff involvement. SPI cannot be imposed top down, nor can it be imposed from the outside. If SPI efforts are to succeed, improvement must be organic—sponsored by technical managers and senior technologists, and adopted by local practitioners.

Process integration and understanding. The software process does not exist in an organizational vacuum. It must be characterized in a manner that is integrated with other business processes and requirements. To accomplish this, those responsible for the SPI effort must have an intimate knowledge and understanding of other business processes. In addition, they must understand the "as is" software process and appreciate how much transitional change is tolerable within the local culture.

A customized SPI strategy. There is no cookie-cutter SPI strategy. As I noted earlier in this chapter, the SPI road map must be adapted to the local environment—team culture, product mix, and local strengths and weaknesses must all be considered.

Solid management of the SPI project. SPI is a project like any other. It involves coordination, scheduling, parallel tasks, deliverables, adaptation (when risks become realities), politics, budget control, and much more. Without active and effective management, an SPI project is doomed to failure.

30.3 THE CMMI

WebRef
Complete information on the CMMI can be obtained at **www.sei.cmu.edu/cmmi/**.

The original CMM was developed and upgraded by the Software Engineering Institute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability Maturity Model Integration* (CMMI) [CMM07], a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model describes a process in two dimensions as illustrated in Figure 30.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

Level 0: *Incomplete*—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

Level 1: *Performed*—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

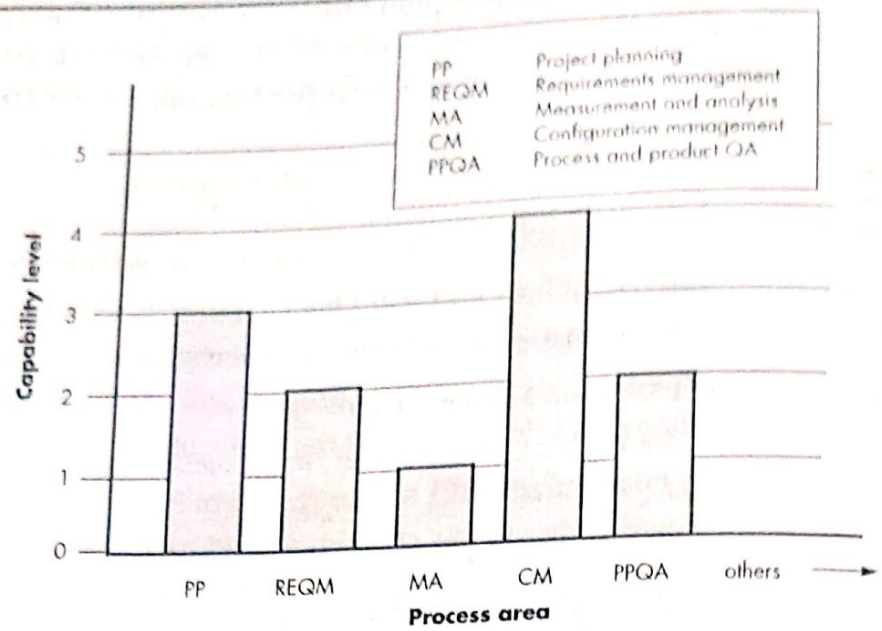

FIGURE 30.2 CMMI process area capability profile. *Source: [Phi02].*

Level 2: *Managed*—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM07].

ADVICE

ery organization ould strive to ieve the intent of CMMI. However, lementing every ct of the model be overkill in your tion.

Level 3: *Defined*—all capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" [CMM07].

Level 4: *Quantitatively managed*—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" [CMM07].

Level 5: *Optimized*—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

For example, **project planning** is one of eight process areas defined by the CMMI for "project management" category.[6] The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are [CMM07]:

SG 1 Establish Estimates

SP 1.1-1 Estimate the Scope of the Project

SP 1.2-1 Establish Estimates of Work Product and Task Attributes

SP 1.3-1 Define Project Life Cycle

SP 1.4-1 Determine Estimates of Effort and Cost

SG 2 Develop a Project Plan

SP 2.1-1 Establish the Budget and Schedule

SP 2.2-1 Identify Project Risks

SP 2.3-1 Plan for Data Management

SP 2.4-1 Plan for Project Resources

SP 2.5-1 Plan for Needed Knowledge and Skills

SP 2.6-1 Plan Stakeholder Involvement

SP 2.7-1 Establish the Project Plan

SG 3 Obtain Commitment to the Plan

SP 3.1-1 Review Plans That Affect the Project

SP 3.2-1 Reconcile Work and Resource Levels

SP 3.3-1 Obtain Plan Commitment

WebRef

Complete information as well as a downloadable version of the CMMI can be obtained at www.sei.cmu.edu/cmmi/.

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved. To illustrate, the generic goals (GG) and practices (GP) for the **project planning** process area are [CMM07]:

GG 1 Achieve Specific Goals

GP 1.1 Perform Base Practices

GG 2 Institutionalize a Managed Process

GP 2.1 Establish an Organizational Policy

GP 2.2 Plan the Process

GP 2.3 Provide Resources

GP 2.4 Assign Responsibility

GP 2.5 Train People

6 Other process areas defined for "project management" include: project monitoring and control, supplier agreement management, integrated project management for IPPD, risk management, integrated teaming, integrated supplier management, and quantitative project management.

GP 2.6 Manage Configurations

GP 2.7 Identify and Involve Relevant Stakeholders

GP 2.8 Monitor and Control the Process

GP 2.9 Objectively Evaluate Adherence

GP 2.10 Review Status with Higher-Level Management

GG 3 Institutionalize a Defined Process

GP 3.1 Establish a Defined Process

GP 3.2 Collect Improvement Information

GG 4 Institutionalize a Quantitatively Managed Process

GP 4.1 Establish Quantitative Objectives for the Process

GP 4.2 Stabilize Subprocess Performance

GG 5 Institutionalize an Optimizing Process

GP 5.1 Ensure Continuous Process Improvement

GP 5.2 Correct Root Causes of Problems

The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The relationship between maturity levels and process areas is shown in Figure 30.3.

INFO

The CMMI—Should We or Shouldn't We?

The CMMI is a process meta-model. It defines (in 700+ pages) the process characteristics that should exist if an organization wants to establish a software process that is complete. The question that has been debated for well over a decade is: "Is the CMMI overkill?" Like most things in life (and in software), the answer is not a simple "yes" or "no."

The spirit of the CMMI should always be adopted. At the risk of oversimplification, it argues that software development must be taken seriously—it must be planned thoroughly, it must be controlled uniformly, it must be tracked accurately, and it must be conducted professionally. It must focus on the needs of project stakeholders, the skills of the software engineers, and the quality of the end product. No one would argue with these ideas.

The detailed requirements of the CMMI should be seriously considered if an organization builds large complex systems that involve dozens or hundreds of people over many months or years. It may be that the CMMI is "just right" in such situations, if the organizational culture is amenable to standard process models and management is committed to making it a success. However, in other situations, the CMMI may simply be too much for an organization to successfully assimilate. Does this mean that the CMMI is "bad" or "overly bureaucratic" or "old fashioned?" No . . . it does not. It simply means that what is right for one organizational culture may not be right for another.

The CMMI is a significant achievement in software engineering. It provides a comprehensive discussion of the activities and actions that should be present when an organization builds computer software. Even if a software organization chooses not to adopt its details, every software team should embrace its spirit and gain insight from its discussion of software engineering process and practice.

FIGURE 30.3 Process areas required to achieve a maturity level. *Source:* [Phi02].

Level	Focus	Process Areas
Optimizing	Continuous process improvement	Organizational innovation and deployment Causal analysis and resolution
Quantitatively managed	Quantitative management	Organizational process performance Quantitative project management
Defined	Process standardization	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Integrated supplier management Risk management Decision analysis and resolution Organizational environment for integration Integrated teaming
Managed	Basic project management	Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management
Performed		

30.4 THE PEOPLE CMM

KEY POINT The People CMM suggests practices that improve the workforce competence and culture.

A software process, no matter how well conceived, will not succeed without talented, motivated software people. The *People Capability Maturity Model* "is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce" [Cur02]. Developed in the mid-1990s and refined over the intervening years, the goal of the People CMM is to encourage continuous improvement of generic workforce knowledge (called "core competencies"), specific software engineering and project management skills (called "workforce competencies"), and process-related abilities.

Like the CMM, CMMI, and related SPI frameworks, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes. These maturity levels [CMM08] are tied to the existence (within an organization) of a set of key process areas (KPAs). An overview of organizational levels and related KPAs is shown in Figure 30.4.

FIGURE 30.4 Process areas for the People CMM

Level	Focus	Process Areas
Optimized	*Continuous improvement*	Continuous workforce innovation Organizational performance alignment Continuous capability improvement
Predictable	*Quantifies and manages knowledge, skills, and abilities*	Mentoring Organizational capability management Quantitative performance management Competency-based assets Empowered workgroups Competency integration
Defined	*Identifies and develops knowledge, skills, and abilities*	Participatory culture Workgroup development Competency-based practices Career development Competency development Workforce planning Competency analysis
Managed	*Repeatable, basic people management practices*	Compensation Training and development Performance management Work environment Communication and co-ordination Staffing
Initial	*Inconsistent practices*	

The People CMM complements any SPI framework by encouraging an organization to nurture and improve its most important asset—its people. As important, it establishes a workforce atmosphere that enables a software organization to "attract, develop, and retain outstanding talent" [CMM08].

30.5 OTHER SPI FRAMEWORKS

Although the SEI's CMM and CMMI are the most widely applied SPI frameworks, a number of alternatives[7] have been proposed and are in use. Among the most widely used of these alternatives are:

- **SPICE**—an international initiative to support ISO's process assessment and life cycle process standards [SPI99]
- **ISO/IEC 15504** for (Software) Process Assessment [ISO08]

- **Bootstrap**—an SPI framework for small and medium-sized organizations that conforms to SPICE [Boo06]
- **PSP and TSP**—individual and team-specific SPI frameworks ([Hum97], [Hum00]) that focus on process in-the-small, a more rigorous approach to software development coupled with measurement
- **TickIT**—an auditing method [Tic05] that assesses an organization's compliance to ISO Standard 9001:2000

A brief overview of each of these SPI frameworks is presented in the paragraphs that follow. If you have further interest, a wide array of print and Web-based resources is available for each.

In addition to the CMM, there other frameworks that we might consider?

SPICE. The SPICE (*Software Process Improvement and Capability dEtermination*) model provides an SPI assessment framework that is compliant with ISO 15504:2003 and ISO 12207. The SPICE document suite [SDS08] presents a complete SPI framework including a model for process management, guidelines for conducting an assessment and rating the process under consideration, construction, selection, and use of assessment instruments and tools, and training for assessors.

Bootstrap. The *Bootstrap* SPI framework "has been developed to ensure conformance with the emerging ISO standard for software process assessment and improvement (SPICE) and to align the methodology with ISO 12207" [Boo06]. The objective of Bootstrap is to evaluate a software process using a set of software engineering best practices as a basis for assessment. Like the CMMI, Bootstrap provides a process maturity level using the results of questionnaires that gather information about the "as is" software process and software projects. SPI guidelines are based on maturity level and organizational goals.

PSP and TSP. Although SPI is generally characterized as an organizational activity, there is no reason why process improvement cannot be conducted at an individual or team level. Both PSP and TSP (Chapter 2) emphasize the need to continuously collect data about the work that is being performed and to use that data to develop strategies for improvement. Watts Humphrey [Hum97], the developer of both methods, comments:

> The PSP [and TSP] will show you how to plan and track your work and how to consistently produce high quality software. Using PSP [and TSP] will give you the data that show the effectiveness of your work and identify your strengths and weaknesses. . . . To have a successful and rewarding career, you need to know your skills and abilities, strive to improve them, and capitalize on your unique talents in the work you do.

Quote:

"Software organizations have exhibited significant shortcomings in their ability to capitalize on the experiences gained from completed projects."

NASA

TickIT. The Ticket auditing method ensures compliance with *ISO 9001:2000 for Software*—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

The underlying strategy suggested by ISO 9001:2000 is described in the following manner [ISO01]:

> ISO 9001:2000 stresses the importance for an organization to identify, implement, manage and continually improve the effectiveness of the processes that are necessary for the quality management system, and to manage the interactions of these processes in order to achieve the organization's objectives. . . . Process effectiveness and efficiency can be assessed through internal or external review processes and be evaluated on a maturity scale.

WebRef
An excellent summary of ISO 9001: 2000 can be found at **http://praxiom.com/iso-9001.htm**.

ISO 9001:2000 has adopted a "plan-do-check-act" cycle that is applied to the quality management elements of a software project. Within a software context, "plan" establishes the process objectives, activities, and tasks necessary to achieve high-quality software and resultant customer satisfaction. "Do" implements the software process (including both framework and umbrella activities). "Check" monitors and measures the process to ensure that all requirements established for quality management have been achieved. "Act" initiates software process improvement activities that continually work to improve the process. TickIt can be used throughout the "plan-do-check-act" cycle to ensure that SPI progress is being made. TickIT auditors assess the application of the cycle as a precursor to ISO 9001:2000 certification. For a detailed discussion of ISO 9001:2000 and TickIT you should examine [Ant06], [Tri05], or [Sch03].

30.6 SPI Return on Investment

SPI is hard work and requires substantial investment of dollars and people. Managers who approve the budget and resources for SPI will invariably ask the question: "How do I know that we'll achieve a reasonable return for the money we're spending?"

At a qualitative level, proponents of SPI argue that an improved software process will lead to improved software quality. They contend that improved process will result in the implementation of better quality filters (resulting in fewer propagated defects), better control of change (resulting in less project chaos), and less technical rework (resulting in lower cost and better time-to-market). But can these qualitative benefits be translated into quantitative results? The classic return on investment (ROI) equation is:

$$\text{ROI} = \left[\frac{\Sigma(\textit{benefits}) - \Sigma(\textit{costs})}{\Sigma(\textit{costs})}\right] \times 100\%$$

where

benefits include the cost savings associated with higher product quality (fewer defects), less rework, reduced effort associated with changes, and the income that accrues from shorter time-to-market.

costs include both direct SPI costs (e.g., training, measurement) and indirect costs associated with greater emphasis on quality control and change management activities and more rigorous application of software engineering methods (e.g., the creation of a design model).

In the real world, these quantitative benefits and costs are sometimes difficult to measure with accuracy, and all are open to interpretation. But that doesn't mean that a software organization should conduct an SPI program without careful analysis of the costs and benefits that accrue. A comprehensive treatment of ROI for SPI can be found in a unique book by David Rico [Ric04].

30.7 SPI Trends

Over the past two decades, many companies have attempted to improve their software engineering practices by applying an SPI framework to effect organizational change and technology transition. As I noted earlier in this chapter, over half fail in this endeavor. Regardless of success or failure, all spend significant amounts of money. David Rico [Ric04] reports that a typical application of an SPI framework such as the SEI CMM can cost between $25,000 and $70,000 per person and take years to complete! It should come as no surprise that the future of SPI should emphasize a less costly and time-consuming approach.

To be effective in the twenty-first century world of software development, future SPI frameworks must become significantly more agile. Rather than an organizational focus (which can take years to complete successfully), contemporary SPI efforts should focus on the project level, working to improve a team process in weeks, not months or years. To achieve meaningful results (even at the project level) in a short time frame, complex framework models may give way to simpler models. Rather than dozens of key practices and hundreds of supplementary practices, an agile SPI framework should emphasize only a few pivotal practices (e.g., analogous to the framework activities discussed throughout this book).

Any attempt at SPI demands a knowledgeable workforce, but education and training expenses can be expensive and should be minimized (and streamlined). Rather than classroom courses (expensive and time consuming), future SPI efforts should rely on Web-based training that is targeted at pivotal practices. Rather than far-reaching attempts to change organizational culture (with all of the political perils that ensue), cultural change should occur as it does in the real world, one small group at a time until a tipping point is reached.

The SPI work of the past two decades has significant merit. The frameworks and models that have been developed represent substantial intellectual assets for the software engineering community. But like all things, these assets guide future attempts at SPI not by becoming a recurring dogma, but by serving as the basis for better, simpler, and more agile SPI models.