26.3 Software Reviews

ADVICE

Reviews are like filters in the software process workflow. Too few, and the flow is "dirty." Too many, and the flow slows to a trickle. Use metrics to determine which reviews work and emphasize them.

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

> Technical work needs reviewing for the same reason that pencils need erasers: *To err is human.* The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else.

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review. In this book, however, we focus on the *formal technical review,* sometimes called a *walkthrough* or an *inspection.* A formal technical review (FTR) is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for uncovering errors and improving software quality.

INFO

Bugs, Errors, and Defects

The goal of SQA is to remove quality problems in the software. These problems are referred to by various names—"bugs," "faults," "errors," or "defects" to name a few. Are each of these terms synonymous, or are there subtle differences between them?

In this book we have made a clear distinction between an *error* (a quality problem found *before* the software is released to end-users) and a *defect* (a quality problem found only *after* the software has been released to end-users[2]). We make this distinction because errors and defects have very different economic, business, psychological, and human impact. As software engineers, we want to find and correct as many errors as possible before the customer and/or end-user encounter them. We want to avoid defects—because defects (justifiably) make software people look bad.

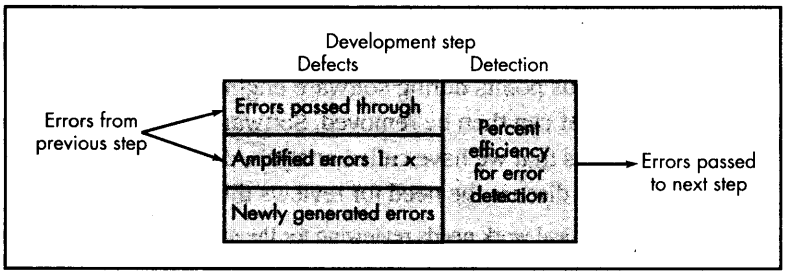
It is important to note, however, that the temporal distinction made between errors and defects in this book is *not* mainstream thinking. The general consensus within the software engineering community is that defects and errors, faults, and bugs are synonymous. That is, the point in time that the problem was encountered has no bearing on the term used to describe the problem. Part of the argument in favor of this view is that it is sometimes difficult to make a clear distinction between pre- and post-release (e.g., consider an incremental process used in agile development [Chapter 4]).

Regardless of how you choose to interpret these terms, recognize that the point in time at which a problem is discovered does matter and that software engineers should try hard—*very* hard—to find problems before their customers and end-users encounter them. If you have further interest in this issue, a reasonably thorough discussion of the terminology surrounding "bugs" can be found at www.softwaredevelopment.ca/bugs.shtml.

2 If software process improvement is considered, a quality problem that is propagated from one process framework activity (e.g., modeling) to another (e.g., construction) can also be called a "defect" (because the problem should have been found before a work product (e.g., a design model) was "released" to the next activity.

FIGURE 26.2

Defect amplification model



26.3.1 Cost Impact of Software Defects

ADVICE

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

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

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

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

26.3.2 Defect Amplification and Removal

A defect amplification model [IBM81] can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of a software engineering process. The model is illustrated schematically in Figure 26.2. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.

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

FIGURE 26.3

Defect amplification—no reviews

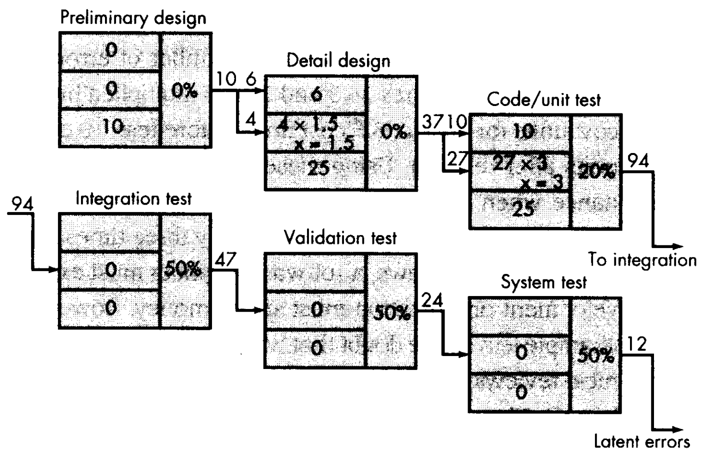
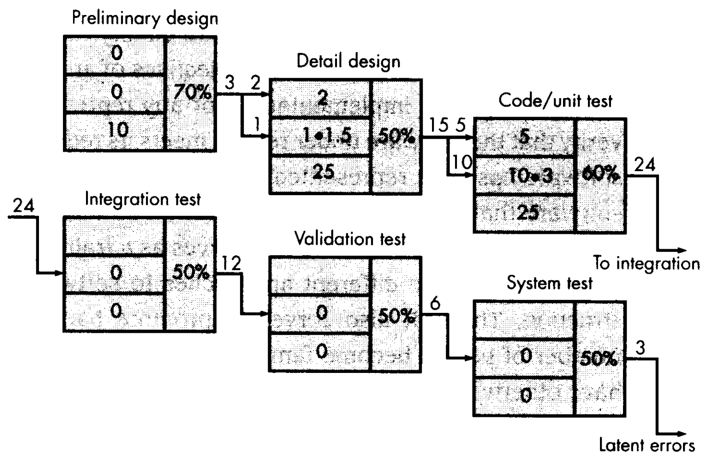


FIGURE 26.4

Defect amplification—reviews conducted



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

Niccolo Machiavelli

Figure 26.3 illustrates a hypothetical example of defect amplification for a software process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent defects are released to the field. Figure 26.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, 10 initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent defects exist. Recalling the relative costs associated with the discovery and correction of errors, overall cost (with and without review for our hypothetical example) can be established. The number of errors uncovered during each of the steps noted in Figures 26.3 and 26.4 is multiplied by the cost to remove an error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release). Using these data, the total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted, total cost is 2177 units—nearly three times more costly.

To conduct reviews, a software engineer must expend time and effort, and the development organization must spend money. However, the results of the preceding example leave little doubt that we can pay now or pay much more later. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

26.4 Formal Technical Reviews

When we conduct FTRs, what are our objectives?

A formal technical review is a software quality control activity performed by software engineers (and others). The objectives of an FTR are (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and construction. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

"There is no urge so great as for one man to edit another man's work."

Mark Twain

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

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

26.4.1 The Review Meeting

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

- Between three and five people (typically) should be involved in the review.

- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

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

KEY POINT
An FTR focuses on a relatively small portion of a work product.

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

ADVICE
In some situations, it's a good idea to have someone other than the producer walk through the product undergoing review. This leads to a literal interpretation of the work product and better error recognition.

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

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

26.4.2 Review Reporting and Record Keeping

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

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

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

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

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

"A meeting is too often an event in which minutes are taken and hours are wasted."

Author unknown

26.4.3 Review Guidelines

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

ADVICE

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

1. *Review the product, not the producer.* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle.
2. *Set an agenda and maintain it.* One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. *Limit debate and rebuttal.* When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. *Enunciate problem areas, but don't attempt to solve every problem noted.* A review is not a problem-solving session. Problem solving should be postponed until after the review meeting.
5. *Take written notes.* It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. *Limit the number of participants and insist upon advance preparation.* Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review

team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).

7. *Develop a checklist for each product that is likely to be reviewed.* A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues.
8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as a task during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.
9. *Conduct meaningful training for all reviewers.* To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews.
10. *Review your early reviews.* Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

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

Ralph Waldo Emerson

Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a software organization should experiment to determine what approach works best in a local context. Porter and his colleagues [POR95] provide excellent guidance for this type of experimentation.

26.4.4 Sample-Driven Reviews

In an ideal setting, every software engineering work product would undergo a formal technical review. In the real world of software projects, resources are limited and time is short. As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized. Thelin and his colleagues [THE01] address this issue when they state:

> Inspections [FTRs] are only viewed efficient if many faults are found during the fault searching part. If many faults are found in the artifacts [work products], the inspections are necessary. If, on the other hand, only few faults are found, the inspection has been a waste of time for several people involved in the inspections[4]. Moreover, software projects which are late often decrease the time for inspection activities, which leads to a lack of

4 Of course, it can be argued that by conducting reviews we encourage producers to focus on quality, even if no errors are found.

quality. A solution would be to prioritize the resources for the inspection activities and thereby concentrate the available resources on the artifacts that are most fault-prone.

ADVICE

Reviews take time, but it's time well spent. However, if time is short and you have no other option, do not dispense with reviews. Rather, use sample-driven reviews.

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

To be effective, the sample driven review process must attempt to quantify those work products that are primary targets for full FTRs. To accomplish this, the following steps are suggested [THE01]:

1. Inspect a fraction a_i of each software work product, *i*. Record the number of faults, f_i found within a_i.
2. Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.
3. Sort the work products in descending order according to the gross estimate of the number of faults in each.
4. Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must (1) be representative of the work product as a whole and (2) large enough to be meaningful to the reviewer(s) who does the sampling. As a_i increases, the likelihood that the sample is a valid representation of the work product also increases. However, the resources required to do sampling also increase. A software engineering team must establish the best value for a_i for the types of work products produced.[5]

SAFEHOME

SQA Issues

The scene: Doug Miller's office as [illegible] project begins.

[illegible] Doug Miller (manager of the *SafeHome* [illegible] engineering team) and other members of the [illegible] engineering team.

[illegible]

[illegible] didn't spend time developing an SQA [illegible] but we're already into it and we [illegible] quality . . . right?

Jamie: Sure. We've already decided that as we develop the requirements model [Chapters 7 and 8], Ed has committed to develop a V&V procedure for each requirement.

Doug: That's really good, but we're not going to wait until testing to evaluate quality, are we?

Vinod: No! Of course not. We've got reviews scheduled into the project plan for this software increment. We'll begin quality control with the reviews.

5 Thelin and his colleagues have conducted a detailed simulation that can assist in making this determination. See [THE01] for details.

Jamie: I'm a bit concerned that we won't have enough time to conduct all the reviews. In fact, I know we won't.

Doug: Hmmm. So what do you propose?

Jamie: I say we select those elements of the analysis and design model that are most critical to *SafeHome* and review them.

Vinod: But what if we miss something in a part of the model we don't review?

Shakira: I read something about a sampling technique [Section 26.4.4] that might help us target candidates for review. (Shakira explains the approach.)

Jamie: Maybe . . . but I'm not sure we even have time to sample every element of the models.

Vinod: What do you want us to do, Doug?

Doug: Let's steal something from Extreme Programming [Chapter 4]. We'll develop the elements of each model in pairs—two people—and conduct an informal review of each as we go. We'll then target "critical" elements for a more formal team review, but keep those reviews to a minimum. That way, everything gets looked at by more than one set of eyes, but we still maintain our delivery dates.

Jamie: That means we're going to have to revise the schedule.

Doug: So be it. Quality trumps schedule on this project.

26.5 Formal Approaches to SQA

Over the past two decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object [SOM01]. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements (Chapter 28) 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 (Chapters 28 and 29) are not new. Dijkstra [DIJ76] and Linger, Mills, and Witt [LIN79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 11).

26.6 Statistical Software Quality Assurance

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

What steps are required to perform statistical SQA?

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).

3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

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

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

Lowell Arthur

26.6.1 A Generic Example

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

- Incomplete or erroneous specifications (IES).
- Misinterpretation of customer communication (MCC).
- Intentional deviation from specifications (IDS).
- Violation of programming standards (VPS).
- Error in data representation (EDR).
- Inconsistent component interface (ICI).
- Error in design logic (EDL).
- Incomplete or erroneous testing (IET).
- Inaccurate or incomplete documentation (IID).
- Error in programming language translation of design (PLT).
- Ambiguous or inconsistent human/computer interface (HCI).
- Miscellaneous (MIS).

To apply statistical SQA, the table in Figure 26.5 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, the software developer might implement facilitated requirements gathering techniques (Chapter 7) to improve the quality of customer communication and specifications. To improve EDR, the developer

FIGURE 26.5
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%

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 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!*

A comprehensive discussion of statistical SQA is beyond the scope of this book. Interested readers should see [GOH02], [SCH98], or [KAN95].

26.6.2 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." [ISI03]. 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, 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.

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

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

- *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements
- *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

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

A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject. The interested reader should see [ISI03], [SNE03], and [PAN00].

26.7 Software Reliability

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

"The unavoidable price of reliability is simplicity."

C.A.R. Hoare

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.

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

There has been debate over the relationship between key concepts in hardware reliability and their applicability to software (e.g., [LIT89], [ROO90]). Although an irrefutable link has yet 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), where

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair,*[6] respectively.

Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP. Stated simply, an end-user is concerned with failures, not with the total error count. Because each 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.

ADVICE
Some aspects of availability (not discussed here) have nothing to do with failure. For example, schedule downtime (for support functions) causes the software to be unavailable.

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

Availability = [MTTF/(MTTF + 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.

26.7.2 Software Safety

Software safety [LEV86] 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

6 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.

process, software design features can be specified that will either eliminate or control potential hazards.

"I cannot imagine any condition which would cause this ship to founder. Modern shipbuilding has gone beyond that."
E. I. Smith, captain of the Titanic

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

- Causes uncontrolled acceleration that cannot be stopped.
- Does not respond to depression of brake pedal (by turning off).
- Does not engage when switch is activated.
- Slowly loses or gains speed.

Once these system-level hazards are identified, analysis techniques are used to assign severity and probability of occurrence.[7] To be effective, software must be analyzed in the context of the entire system. For example, a subtle user input error (people are system components) may be magnified by a software fault to produce control data that improperly positions a mechanical device. If a set of external environmental conditions are met (and only if they are met), the improper position of the mechanical device will cause a disastrous failure. Analysis techniques such as fault tree analysis [VES81], real-time logic [JAN86], or Petri net models [LEV87] can be used to predict the chain of events that can cause hazards and the probability that each of the events will occur to create the chain.

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

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. Those readers with further interest should refer to Leveson's [LEV95] book on the subject.

7 This approach is similar to the risk analysis methods described in Chapter 25. The primary difference is the emphasis on technology issues rather than project related topics.

26.8 The ISO 9000 Quality Standards[8]

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. ISO 9000 describes a quality assurance system in generic terms that can be applied to any business regardless of the products or services offered.

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

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.

ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interpret the standard for use in the software process.

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

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. 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. For further information on ISO 9001, the interested reader should see [HOY02], [GAA01], or [CIA01].

INFO

The ISO 9001:2000 Standard

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 recordkeeping.

Support quality control and assurance.

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

- 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, 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.

26.9 The SQA Plan

The *SQA Plan* provides a road map for instituting software quality assurance. Developed by the SQA group (or the software team if a 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 [IEE94]. 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 procedures (Chapter 27) for managing change; (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 13 and 14) are often included within the SQA tools category.

Representative Tools[9]

ARM, developed by NASA (satc.gsfc.nasa.gov/tools/index.html), provides

9 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.

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 Cookbook, developed by Systma and Manley (www.sytsma.com/tqmtools/tqmtoolmenu.html), provides useful descriptions of classic quality management tools such as control charts, scatter diagrams, affinity diagrams, and matrix diagrams.

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

TQM Tools, developed by Bain & Company (www.bain.com), provide useful descriptions of a variety of management tools used for TQM and related quality management methods.

26.10 Summary

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

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

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

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

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

References

[ALV64] Alvin, W. H., von (ed.), *Reliability Engineering,* Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology,* 1987.

[ART92] Arthur, L. J., *Improving Software Quality: An Insider's Guide to TQM,* Wiley, 1992.

[ART97] Arthur, L. J., "Quantum Improvements in Software System Quality, *CACM,* vol. 40, no. 6, June 1997, pp. 47-52.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981.

[CIA01] Cianfrani, C. A., et al., *ISO 9001:2000 Explained,* 2nd ed., American Society for Quality, 2001.

[CRO79] Crosby, P., *Quality Is Free,* McGraw-Hill, 1979.

[DEM86] Deming, W. E., *Out of the Crisis,* MIT Press, 1986.

[DEM99] DeMarco, T., "Management Can Make Quality (Im)possible," Cutter IT Summit, Boston, April 1999.

[DIJ76] Dijkstra, E., *A Discipline of Programming,* Prentice-Hall, 1976.

[DUN82] Dunn, R., and R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982.

[FRE90] Freedman, D. P., and G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews,* 3rd ed., Dorset House, 1990.

[GAA01] Gaal, A., *ISO 9001:2000 for Small Business,* Saint Lucie Press, 2001.

[GIL93] Gilb, T., and D. Graham, *Software Inspections,* Addison-Wesley, 1993.

[GLA98] Glass, R., "Defining Quality Intuitively," *IEEE Software,* May 1998, pp. 103- 104, 107.

[GOH02] Goh, T., V. Kuralmani, and M. Xie, *Statistical Models and Control Charts for High Quality Processes,* Kluwer Academic Publishers, 2002.

[HOY02] Hoyle, D., *ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach,* 4th ed., Butterworth-Heinemann, 2002.

[IBM81] "Implementing Software Inspections," course notes, IBM Systems Sciences Institute, IBM Corporation, 1981.

[IEE94] *Software Engineering Standards,* 1994, IEEE Computer Society, 1994.

[ISI03] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experienced Quality Practitioners," 2003, available at http://www.isixsigma.com/library/content/six-sigma-newbie.asp.

[JAN86] Jahanian, F., and A. K. Mok, "Safety Analysis of Timing Properties of Real-Time Systems," *IEEE Trans. Software Engineering,* vol. SE-12, no. 9, September 1986, pp. 890-904.

[JON86] Jones, T. C., *Programming Productivity,* McGraw-Hill, 1986.

[KAN95] Kan, S. H., *Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995.

[LEV86] Leveson, N. G., "Software Safety: Why, What, and How," *ACM Computing Surveys,* vol. 18, no. 2, June 1986, pp. 125-163.

[LEV87] Leveson, N. G., and J. L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Trans. Software Engineering,* vol. SE-13, no. 3, March 1987, pp. 386-397.

[LEV95] Leveson, N. G., *Safeware: System Safety and Computers,* Addison-Wesley, 1995.

[LIN79] Linger, R., H. Mills, and B. Witt, *Structured Programming,* Addison-Wesley, 1979.

[LIT89] Littlewood, B., "Forecasting Software Reliability," in *Software Reliability: Modeling and Identification,* (S. Bittanti, ed.), Springer-Verlag, 1989, pp. 141-209.

[MUS87] Musa, J. D., A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures,* McGraw-Hill, 1987.

[PAN00] Pande, P., et al., *The Six Sigma Way,* McGraw-Hill, 2000.

[POR95] Porter, A., H. Siy, C. A. Toman, and L. G. Votta, "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Washington, D.C., October 1995, ACM Press, pp. 92-103.

[ROO90] Rook, J., *Software Reliability Handbook,* Elsevier, 1990.

[SCH98] Schulmeyer, G. C., and J. I. McManus (eds.), *Handbook of Software Quality Assurance,* 3rd ed., Prentice-Hall, 1998.

[SOM01] Somerville, I., *Software Engineering,* 6th ed., Addison-Wesley, 2001.

[SNE03] Snee, R., and R. Hoerl, *Leading Six Sigma,* Prentice-Hall, 2003.

[THE01] Thelin, T., H. Petersson, and C. Wohlin, "Sample Driven Inspections," *Proceedings Workshop on Inspection in Software Engineering (WISE'01),* Paris, France, July 2001, pp. 81-91, can be downloaded from http://www.cas.mcmaster.ca/ wise/wise01/ThelinPetersson-Wohlin.pdf.

[VES81] Veseley, W. E., et al., *Fault Tree Handbook,* U.S. Nuclear Regulatory Commission, NUREG-0492, January 1981.

Problems and Points to Ponder

26.1. Some people argue that an FTR should assess programming style as well as correctness. Is this a good idea? Why?

26.2. Is it possible to assess the quality of software if the customer keeps changing what it is supposed to do?

26.3. Acquire a copy of ISO 9001:2000 and ISO 9000-3. Prepare a presentation that discusses three ISO 9001 requirements and how they apply in a software context.

26.4. Early in this chapter we noted that "variation control is the heart of quality control." Since every program that is created is different from every other program, what are the variations that we look for and how do we control them?

26.5. Can a program be correct and still not exhibit good quality? Explain.

26.6. Why is there often tension between a software engineering group and an independent software quality assurance group? Is this healthy?

26.7. You have been given the responsibility for improving the quality of software across your organization. What is the first thing that you should do? What's next?

26.8. The MTBF concept for software is open to criticism. Can you think of a few reasons why?

26.9. A formal technical review is effective only if everyone has prepared in advance. How do you recognize a review participant who has not prepared? What do you do if you're the review leader?

26.10. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

26.11. Consider two safety critical systems that are controlled by computers. List at least three hazards for each that can be directly linked to software failures.

26.12. Research the literature on software reliability, and write a paper that describes one software reliability model. Be sure to provide an example.

26.13. Review the table presented in Figure 26.5 and select four vital few causes of serious and moderate errors. Suggest corrective actions using information presented in other chapters.

26.14. Besides counting errors and defects, are there other countable characteristics of software that imply quality? What are they, and can they be measured directly?

26.15. Can a program be correct and still not be reliable? Explain.

Further Readings and Information Sources

Books by Moriguchi (*Software Excellence: A Total Quality Management Guide,* Productivity Press, 1997) and Horch (*Practical Guide to Software Quality Management,* Artech Publishing, 1996) are excellent management-level presentations on the benefits of formal quality assurance programs for computer software. Books by Deming [DEM86], Juran (*Juran on Quality by Design,* Free Press, 1992), and Crosby ([CRO79] and *Quality Is Still Free,* McGraw-Hill, 1995) do not focus on software, but are must reading for senior managers with software development responsibility. Gluckman and Roome (*Everyday Heroes of the Quality Movement,* Dorset House, 1993) humanizes quality issues by telling the story of the players in the quality process. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 1995) presents a quantitative view of software quality.

The ISO 9001:2000 quality standard is discussed by Cianfani and his colleagues (*ISO 9001:2000 Explained,* second edition, American Society for Quality, 2001) and Gaal (*ISO 9001:2000 for Small Business: Implementing Process-Approach Quality Management,* St. Lucie Press, 2001). Tingley (*Comparing ISO 9000, Malcolm Baldrige, and the SEI CMM* for Software, Prentice-Hall, 1996) provides useful guidance for organizations that are striving to improve their quality management processes.

Books by George (*Lean Six Sigma,* McGraw-Hill, 2002), Pande and his colleagues (*The Six Sigma Way Fieldbook,* McGraw-Hill, 2001), and Pyzdek (*The Six Sigma Handbook,* McGraw-Hill, 2000) describe Six Sigma, a statistical quality management technique that leads to products that have very low defect rates.

Radice (*High Quality, Low Cost Software Inspections,* Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide,* Addison-Wesley, 2001), Gilb and Graham (*Software Inspection,* Addison-Wesley, 1993) and Freedman and Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews,* Dorset House, 1990) provide worthwhile guidelines for conducting effective formal technical reviews.

Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw-Hill, 1998) has written a practical guide to applied software reliability techniques. Anthologies of important papers on software reliability have been edited by Kapur et al. (*Contributions to Hardware and Software Reliability Modelling,* World Scientific Publishing Co., 1999), Gritzalis (*Reliability, Quality and Safety of Software-Intensive Systems,* Kluwer Academic Publishers, 1997), and Lyu (*Handbook of Software Reliability Engineering,* McGraw-Hill, 1996).

Hermann (*Software Safety and Reliability,* Wiley-IEEE Press, 2000), Storey (*Safety-Critical Computer Systems,* Addison-Wesley, 1996) and Leveson [LEV95] continue to be the most comprehensive discussions of software safety published to date. In addition, van der Meulen (*Definitions for Hardware and Software Safety Engineers,* Springer-Verlag, 2000) offers a complete compendium of important concepts and terms for reliability and safety. Gartner (*Testing Safety-Related Software,* Springer-Verlag, 1999) provides specialized guidance for testing safety critical systems. Friedman and Voas (*Software Assessment: Reliability Safety and Testability,* Wiley, 1995) provide useful models for assessing reliability and safety.

A wide variety of information sources on software quality management is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site: **http://www.mhhe.com/pressman.**

Start

CHAPTER 27

CHANGE MANAGEMENT

KEY CONCEPTS
audits
baselines
change
change control
configuration objects
content management
CVS
identification
repository
SCIs
SCM process
standards
status reporting
version control
WebApp CM

Change is inevitable when computer software is built. And change increases the level of confusion among software engineers who are working on a project. Confusion arises when changes are not analyzed before they are made, recorded before they are implemented, reported to those with a need to know, or controlled in a manner that will improve quality and reduce error. Babich [BAB86] discusses this when he states:

> The art of coordinating software development to minimize . . . confusion is called configuration management. Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

Change management, more commonly called *software configuration management* (SCM or CM), is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest.

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that are initiated when a software engineering project begins and terminate only when the software is taken out of operation.

QUICK LOOK

What is it? When you build computer software, change happens. And because it happens, you need to manage it effectively. Change management, also called software configuration management (SCM), is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Who does it? Everyone involved in the software process is involved with change management to some extent, but specialized support positions are sometimes created to manage the SCM process.

Why is it important? If you don't control change, it controls you. And that's never good. It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. For that reason, change management is an essential part of good project management and solid software engineering practice.

What are the steps? Because many work products are produced when software is built, each must be [illegible] plished, m[illegible] trol can be [illegible] maintained [illegible] changes are made [illegible] audited, and to ensure that those with a [illegible] know are informed about changes, reporting [illegible] conducted.

What is the work product? A *Software Configuration Management Plan* defines the project strategy for change management. In addition, when formal SCM is invoked, the change control process produces software change requests, reports, and engineering change orders.

How do I ensure that I've done it right? When every work product can be accounted for, [illegible] and controlled; when every change can be [illegible] and analyzed; when everyone who needs to know about a change has been informed—[illegible] it right.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter, we discuss the specific actions that enable us to manage change.

27.1 Software Configuration Management

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms); (2) work products that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration.*

If each configuration item simply led to other items, little confusion would result. Unfortunately, another variable enters the process—*change.* Change may occur at any time, for any reason. In fact, the First Law of System Engineering [BER80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

"There is nothing permanent except change."

Heraclitus, 500 B.C.

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

What is the origin of changes that are requested for software?

- New business or market conditions dictate changes in product requirements or business rules.
- New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.

- Budgetary or scheduling constraints cause a redefinition of the system or product.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

27.1.1 A SCM Scenario[1]

A typical CM operational scenario involves a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product. In the scenario, assume that the product is a small one involving about 15,000 lines of code being developed by a team of six people. (Note that other scenarios of smaller or larger teams are possible but, in essence, there are generic issues that each of these projects face concerning CM.)

At the operational level, the scenario involves various roles and tasks. For the project manager, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

What are the goals of and the activities performed by each of the constituencies involved in change management?

The goals of the configuration manager are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating them (via a Change Control Board that is responsible for approving changes to the software system), and for authorizing changes. The manager creates and disseminates task lists for the engineers and basically creates the project context. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

For the software engineers, the goal is to work effectively. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting documents. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying

1 This section is extracted from [DAR01]. Special permission to reproduce "Spectrum of Functionality in CM Systems by Susan Dart [DAR01], © 2001 by Carnegie Mellon University is granted by the Software Engineering Institute.

KEY POINT
There must be a mechanism to ensure that simultaneous changes to the same component are properly tracked, managed, and executed.

one another about tasks required and tasks completed. Changes are propagated across each other's work by merging files. Mechanisms exist to ensure that, for components which undergo simultaneous changes, there is some way of resolving conflicts and merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed. The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

The customer uses the product. Since the product is under CM control, the customer follows formal procedures for requesting changes and for indicating bugs in the product.

Ideally, a CM system used in this scenario should support all these roles and tasks; that is, the roles determine the functionality required of a CM system. The project manager sees CM as an auditing mechanism; the configuration manager sees it as a controlling, tracking, and policy making mechanism; the software engineer sees it as a changing, building, and access control mechanism; and the customer sees it as a quality assurance mechanism.

27.1.2 Elements of a Configuration Management System

In her comprehensive white-paper on software configuration management, Susan Dart [DAR01] identifies four important elements that should exist when a configuration management system is developed:

- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enable access to and management of each software configuration item.
- *Process elements*—a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) has been assembled.
- *Human elements*—to implement effective SCM, the software team uses a set of tools and process features (encompassing other CM elements).

These elements (to be discussed in more detail in later sections) are not mutually exclusive. For example, component elements work in conjunction with construction elements as the software process evolves. Process elements guide many human activities that are related to SCM and might therefore be considered human elements as well.

27.1.3 Baselines

ADVICE

Most software changes are justified, so there's no point in complaining about them. Rather, be certain that you have mechanisms in place to handle them.

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple. As time passes, all constituencies know more (about what they need, which approach would be best, how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: *Most changes are justified!*

A *baseline* is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

KEY POINT

A software engineering work product becomes a baseline only after it has been reviewed and approved.

Before a software configuration item becomes a baseline, change may be made quickly and informally. However, once a baseline is established, we figuratively pass through a swinging one-way door. Changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a formal technical review (Chapter 26). For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 27.1.

ADVICE

Be sure that the project database is maintained in a centralized, controlled location.

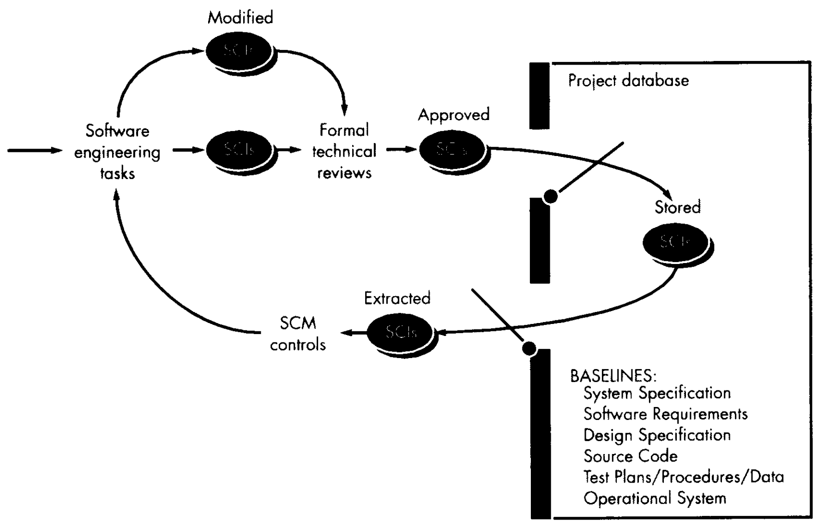
The progression of events that lead to a baseline is also illustrated in Figure 27.1. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository* and discussed in Section 27.2). When a member of a software team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private workspace. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 27.1 illustrate the modification path for a baselined SCI.

27.1.4 Software Configuration Items

A software configuration item is information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of

FIGURE 27.1

Baselined SCIs and the project database



a large specification or one test case in a large suite of tests. More realistically, a SCI is a document, a entire suite of test cases, or a named program component (e.g., a C++ function or a Java applet).

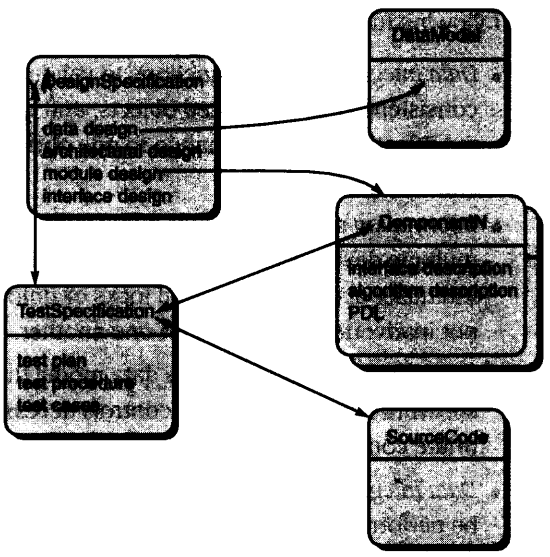
In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, browsers, and other automated tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A *configuration object* has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 27.2, the configuration objects, **DesignSpecification, DataModel, ComponentN, SourceCode** and **TestSpecification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **DataModel** and **ComponentN** are part of the object **DesignSpecification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **SourceCode** object, the interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.[2]

2 These relationships are defined within the database. The structure of the database (repository) is discussed in greater detail in Section 27.2.

FIGURE 27.2

Configuration objects



27.2 The SCM Repository

In the early days of software engineering, software configuration items were maintained as paper documents (or punched computer cards!), placed in file folders or three-ring binders, and stored in metal cabinets. This approach was problematic for many reasons: (1) finding a configuration item when it was needed was often difficult; (2) determining which items were changed, when and by whom was often challenging; (3) constructing a new version of an existing program was time consuming and error-prone; (4) describing detailed or complex relationships between configuration items was virtually impossible.

Today, SCIs are maintained in a project database or repository. Webster's Dictionary defines the word *repository* as "any thing or person thought of as a center of accumulation or storage." During the early history of software engineering, the repository was indeed a person—the programmer who had to remember the location of all information relevant to a software project, who had to recall information that was never written down, and reconstruct information that had been lost. Sadly, using a person as "the center for accumulation and storage" (although it conforms to Webster's definition) does not work very well. Today, the repository is a "thing"—a database that acts as the center for both accumulation and storage of software engineering information. The role of the person (the software engineer) is to interact with the repository using tools that are integrated with it.

27.2.1 The Role of the Repository

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a database management system, but in addition, the repository performs or precipitates the following functions [FOR89]:

What functions are implemented by a SCM repository?

- *Data integrity* includes functions to validate entries to the repository, ensure consistency among related objects, and automatically perform "cascading" modifications when a change to one object demands some change to objects related to it.
- *Information sharing* provides a mechanism for sharing information among multiple developers and between multiple tools, manages and controls multiuser access to data, and locks or unlocks objects so that changes are not inadvertently overlaid on one another.
- *Tool integration* establishes a data model that can be accessed by many software engineering tools, controls access to the data, and performs appropriate configuration management functions.
- *Data integration* provides database functions that allow various SCM tasks to be performed on one or more SCIs.
- *Methodology enforcement* defines an entity-relationship model stored in the repository that implies a specific process model for software engineering; at a minimum, the relationships and objects define a set of steps that must be conducted to build the contents of the repository.
- *Document standardization* is the definition of objects in the database that leads directly to a standard approach for the creation of software engineering documents.

To achieve these functions, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs. For further information, the interested reader should see [SHA95] and [GRI95].

27.2.2 General Features and Content

WebRef
Examples of commercially available repositories can be obtained at **www.software.hp.com/products/SCMGR or otn.oracle.com/documentation/repository.html.**

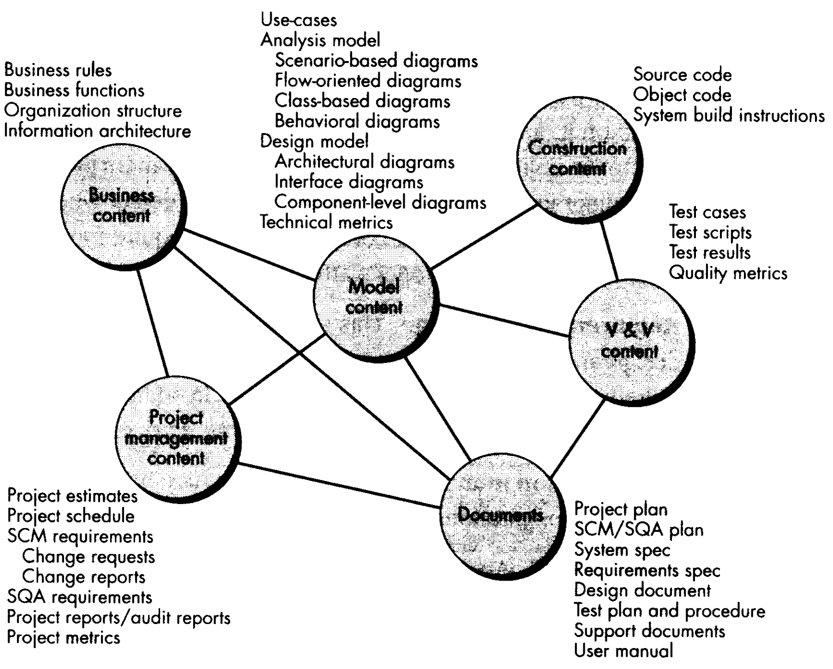
The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. A detailed breakdown of types of representations, documents, and work products that are stored in the repository is presented in Figure 27.3.

A robust repository provides two different classes of services: (1) the same types of services that might be expected from any sophisticated database management system and (2) services that are specific to the software engineering environment.

A repository that serves a software engineering team should (1) integrate with or directly support process management functions; (2) support specific rules that govern the SCM function and the data maintained within the repository; (3) provide an

FIGURE 27.3

Content of the repository



interface to other software engineering tools; and (4) accommodate storage of sophisticated data objects (e.g., text, graphics, video, audio).

27.2.3 SCM Features

To support SCM, the repository must have a tool set that provides support for the following features:

KEY POINT
The repository must be capable of maintaining SCIs related to many different versions of the software. More important, it must provide the mechanisms for assembling these SCIs into a version-specific configuration.

Versioning. As a project progresses, many versions (section 27.3.2) of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.

The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects like screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

Dependency tracking and change management. The repository manages a wide variety of relationships among the configuration objects stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and other work products, and so on. Some of these relationships are merely associations, and some are dependencies or mandatory relationships.

The ability to keep track of all of these relationships is crucial to the integrity of the information stored in the repository and to the generation of work products based on it, and it is one of the most important contributions of the repository concept to the improvement of the software development process. For example, if a UML class diagram is modified, the repository can detect whether related classes, interface definitions, and code components also require modification and can bring affected SCIs to the developer's attention.

Requirements tracing. This special function provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, it provides the ability to identify which requirement generated any given work product (backward tracing).

Configuration management. A configuration management facility keeps track of a series of configurations representing specific project milestones or production releases.

Audit trails. An audit trail establishes additional information about when, why, and by whom changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository.

27.3 The SCM Process

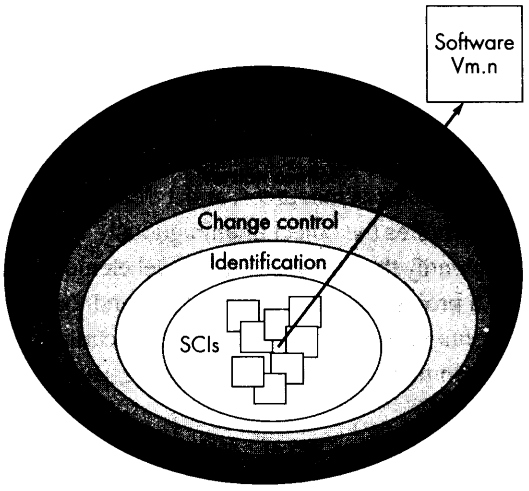
The software configuration management process defines a series of tasks that have four primary objectives: (1) to identify all items that collectively define the software configuration; (2) to manage changes to one or more of these items; (3) to facilitate the construction of different versions of an application; and (4) to ensure that software quality is maintained as the configuration evolves over time.

A process that achieves these objectives need not be bureaucratic and ponderous, but it must be characterized in a manner that enables a software team to develop answers to a set of complex questions:

What questions should the SCM process be designed to answer?

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

FIGURE 27.4
Layers of the SCM process



These questions lead us to the definition of five SCM tasks—identification, version control, change control, configuration auditing, and reporting—illustrated in Figure 27.4.

Referring to the figure, SCM tasks can be viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system. As an SCI moves through a layer, the actions implied by each SCM process layer may or may not be applicable. For example, when a new SCI is created, it must be identified. However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play). A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know. In the sections that follow, we examine each of these SCM process layers in more detail.

27.3.1 Identification of Objects in the Software Configuration

To control and manage software configuration items, each should be separately named and then organized using an object-oriented approach. Two types of objects can be identified [CHO89]: basic objects and aggregate objects.[3] A *basic object* is a unit of information that has been created by a software engineer during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, part of a design model, source code for a component,

3 The concept of an aggregate object [GUS89] has been proposed as a mechanism for representing a complete version of a software configuration.

or a suite of test cases that are used to exercise the code. An *aggregate object* is a collection of basic objects and other aggregate objects. Referring to Figure 27.2, **DesignSpecification** is an aggregate object. Conceptually, it can be viewed as a named (identified) list of pointers that specify basic objects such as **DataModel** and **ComponentN.**

Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify the SCI type (e.g., model element, program, data) represented by the object, a project identifier, and change and/or version information.

KEY POINT
The interrelationships established for configuration objects allow a software engineer to assess the impact of change.

Configuration object identification can also consider the relationships that exist between named objects. For example, using the simple notation

Class diagram <part-of> analysis model;
Analysis model <part-of> requirements specification;

we create a hierarchy of SCIs.

In many cases, objects are interrelated across branches of the object hierarchy. These cross structural relationships can be represented in the following manner:

ADVICE
Even if the project database provides the ability to establish these relationships, they are time-consuming to establish and difficult to keep up-to-date. Although very useful for impact analysis, they are not essential for overall change mangagement.

data model <interrelated>data flow model;
data model <interrelated>test case class m;

In the first case, the interrelationship is between a composite object, while the second relationship is between an aggregate object (**DataModel**) and a basic object (**TestCaseClassM**).

The identification scheme for configuration objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent.

27.3.2 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities: (1) a project database (repository) that stores all relevant configuration objects; (2) a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions); (3) a *make facility* that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software. In addition, version control and change control systems often implement an *issues tracking* (also called *bug tracking*) capa-

KEY POINT
A "make" facility enables a software engineer to extract all relevant configuration objects and construct a specific version of the software.

bility that enables the team to record and track the status of all outstanding issues associated with each configuration object.

"Any change, even a change for the better, is accompanied by drawbacks and discomforts."

Arnold Bennett

A number of version control systems establish a *change set*—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software. Dart [DAR91] notes that a change set "captures all changes to all files in the configuration along with the reason for changes and details of who made the changes and when."

A number of named change sets can be identified for an application or system. This enables a software engineer to construct a version of the software by specifying the change sets (by name) that must be applied to the baseline configuration. To accomplish this, a *system modeling* approach is applied. The system model contains: (1) a *template* that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed, (2) construction rules, and (3) verification rules.[4]

A number of different automated approaches to version control have been proposed over the last two decades. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

SOFTWARE TOOLS

The Concurrent Versions System (CVS)

The use of tools to achieve version control is essential for effective change management. The *Concurrent Versions System* (CVS) is a widely used tool for version control. Originally designed for source code, but useful for any text-based file, the CVS system (1) establishes a simple repository, (2) maintains all versions of a file in a single named file by storing only the differences between progressive versions of the original file, and (3) protects against simultaneous changes to a file by establishing different directories for each developer, thus insulating one from another. CVS merges changes when each developer completes her work.

It is important to note that CVS is not a "build" system; that is, it does not construct a specific version of the software. Other tools (e.g., *Makefile*) must be integrated with CVS to accomplish this. CVS does not implement a change control process (e.g., change requests, change reports, bug tracking).

Even with these limitations, CVS "is a dominant open-source network-transparent version control system [that] is useful for everyone from individual developers to large, distributed teams" [CVS02]. Its client/server architecture allows users to access files via Internet connections, and its open source philosophy makes it available on most popular platforms.

CVS is available at no cost for Windows, Macintosh, and UNIX environments. See www.cvshome.org for further details.

4 It is also possible to query the system model to assess how a change in one component will impact other components.

27.3.3 Change Control

The reality of change control in a modern software engineering context has been summed up beautifully by James Bach [BAC98]:

> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

Bach recognizes that we face a balancing act. Too much change control, and we create problems. Too little, and we create other problems.

"The art of progress is to preserve order amid change and to preserve change amid order."
Alfred North Whitehead

KEY POINT *It should be noted that a number of change requests may be combined to result in a single ECO and that ECOs typically result in changes to multiple configuration objects.*

For a large software engineering project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools. The change control process is illustrated schematically in Figure 27.5. A *change request* is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report,* which is used by a *change control authority* (CCA)—a person or group who makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit.

The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made. As an alternative, the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied. The object(s) is (are) then "checked in" to the database and appropriate version control mechanisms (Section 27.3.2) are used to create the next version of the software.

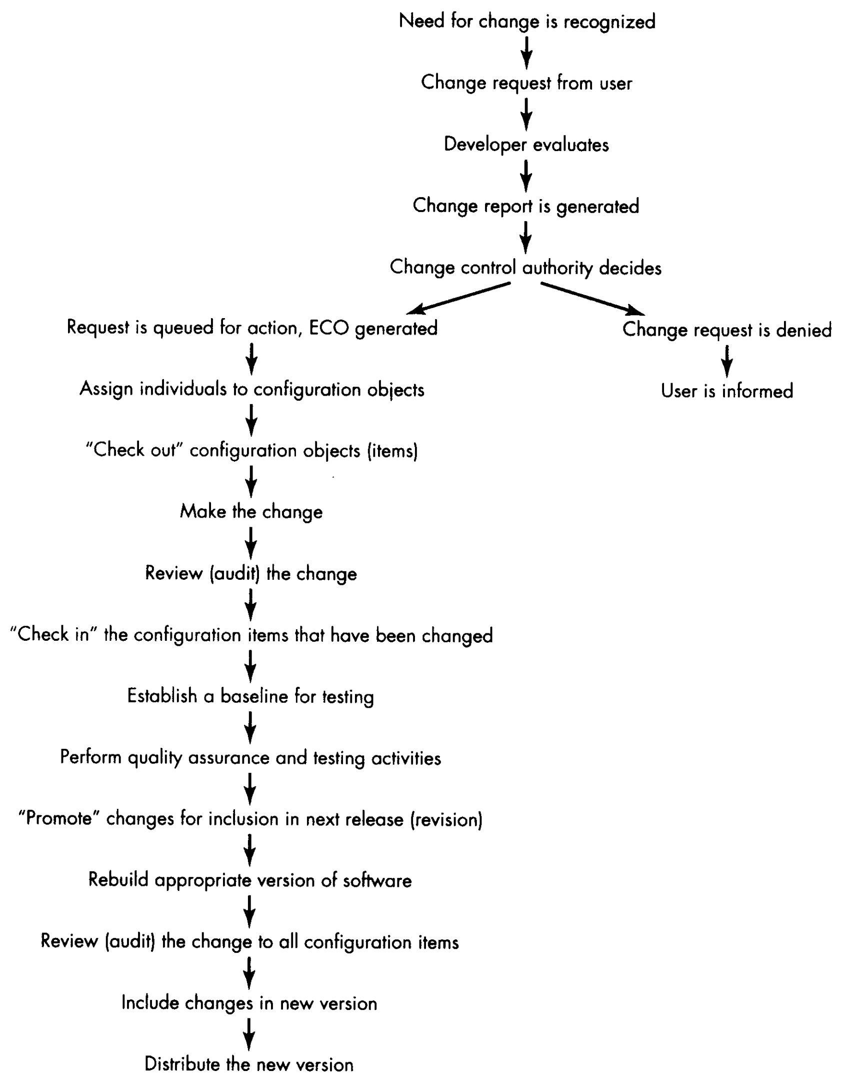
ADVICE *Confusion leads to errors—some of them very serious. Access and synchronization control avoid confusion. Use version and change control tools that implement both.*

These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another [HAR89].

Some readers may begin to feel uncomfortable with the level of bureaucracy implied by the change control process description shown in Figure 27.5. This feeling is

FIGURE 27.5

The change control process


Need for change is recognized
Change request from user
Developer evaluates
Change report is generated
Change control authority decides
Request is queued for action, ECO generated
Change request is denied
User is informed
Assign individuals to configuration objects
"Check out" configuration objects (items)
Make the change
Review (audit) the change
"Check in" the configuration items that have been changed
Establish a baseline for testing
Perform quality assurance and testing activities
"Promote" changes for inclusion in next release (revision)
Rebuild appropriate version of software
Review (audit) the change to all configuration items
Include changes in new version
Distribute the new version

ADVICE

Opt for a bit more change control than you think you'll need. It's likely that too much will be the right amount.

not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately, many have none) have created a number of layers of control to help avoid the problems alluded to here.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone formal technical review and has been approved, a baseline can be created.[5] Once a SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, formal generation of change requests, change reports, and ECOs is dispensed with. However, assessment of each change is conducted, and all changes are tracked and reviewed.

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 27.5.

"Change is inevitable, except for vending machines."

Bumper sticker

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify the customer's perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

SafeHome

SCM Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I know it's early, but we've got to talk about change management.

Vinod (laughing): Hardly. Marketing called this morning with a few "second thoughts." Nothing major, but it's just the beginning.

Jamie: We've been pretty informal about change management on past projects.

Doug: I know, but this is bigger and more visible, and as I recall . . .

Vinod (nodding): We got killed by uncontrolled changes on the home lighting control project . . . remember the delays that . . .

Doug (frowning): A nightmare that I'd prefer not to relive.

Jamie: So what do we do.

Doug: As I see it, three things. First we have to develop—or borrow—a change control process.

Jamie: You mean how people request changes?

Vinod: Yeah, but also how we evaluate the change, decide when to do it (if that's what we decide), and how we keep records of what's affected by the change.

5 A baseline can be created for other reasons as well. For example, when "daily builds" are created, all components checked in by a given time become the baseline for the next day's work.

Doug: Second, we've got to get a really good SCM tool for change and version control.

Jamie: We can build a database for all of our work products.

Vinod: They're called SCIs in this context, and most good tools provide some support for that.

Doug: That's a good start, now we have to . . .

Jamie: Uh, Doug, you said there were three things . . .

Doug (smiling): Third—we've all got to commit to follow the change management process and use the tools—no matter what, okay?

27.3.4 Configuration Audit

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review (presented in detail in Chapter 26) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A formal technical review should be conducted for all but the most trivial changes.

A *software configuration audit* complements the formal technical review by addressing the following questions:

? What are the primary questions that we ask during a configuration audit?

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Has the software process been followed, and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group. Such formal configuration audits also ensure that the correct SCIs (by version) have been incorporated into a specific build and that all documentation is up-to-date and consistent with the version that has been built.

ADVICE

Develop a "need to know" list for every configuration object and keep it up-to-date. When a change is made, be sure that everyone on the list is notified.

27.3.5 Status Reporting

Configuration status reporting (sometimes called *status accounting*) is a SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 27.5. Each time a SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an on-line database or Web site, so that software developers or maintainers can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

SOFTWARE TOOLS

SCM Support

Objective: SCM tools provide support to one or more of the process activities discussed in Section 27.3

Mechanics: Most modern SCM tools work in conjunction with a repository (a database system) and provide mechanisms for identification, version and change control, auditing, and reporting.

Representative Tools[6]

CCC/Harvest, distributed by Computer Associates (www.cai.com), is a multiplatform SCM system.

ClearCase, developed by Rational (www.rational.com), provides a family of SCM functions.

Concurrent Versions System (CVS), an open source tool (www.cvshome.org), is one of the industry's most widely used version control systems (see earlier sidebar).

PVCS, distributed by Merant (www.merant.com), provides a full set of SCM tools that are applicable for both conventional software and WebApps.

SourceForge, distributed by VA Software (sourceforge.net), provides version management, build capabilities, issue/bug tracking, and many other management features.

SurroundSCM, developed by Seapine Software (www.seapine.com), provides complete change management capabilities.

Vesta, distributed by Compac (www.vestasys.org), is a public domain SCM system that can support both small (<10 KLOC) and large (10,000 KLOC) projects.

A comprehensive list of commercial SCM tools and environments can be found at www.cmtoday.com/yp/commercial.html.

27.4 Configuration Management for Web Engineering

In Part 3 of this book, we discussed the special nature of Web applications and the Web engineering process that is required to build them. Among the many characteristics that differentiate WebApps from conventional software is the ubiquitous nature of change.

Web engineering uses an iterative, incremental process model (Chapter 16) that applies many principles derived from agile software development (Chapter 4). Using this

6 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.

approach, an engineering team often develops a WebApp increment in a very short time period via a customer-driven approach. Subsequent increments add additional content and functionality, and each is likely to implement changes that lead to enhanced content, better usability, improved aesthetics, better navigation, enhanced performance, and stronger security. Therefore, in the agile world of Web engineering, change is viewed somewhat differently.

Web engineers must embrace change, and yet a typical agile team eschews all things that appear to be process-heavy, bureaucratic, and formal. Software configuration management is often viewed (albeit incorrectly) to have these characteristics. This seeming contradiction is remedied not by rejecting SCM principles, practices, and tools, but rather by molding them to meet the special needs of Web engineering projects.

27.4.1 Configuration Management Issues for WebApps

What impact does uncontrolled change have on a WebApp?

As WebApps become increasingly important to business survival and growth, the need for configuration management grows. Why? Because without effective controls, improper changes to a WebApp (recall that immediacy and continuous evolution are the dominant attributes of many WebApps) can lead to unauthorized posting of new product information; erroneous or poorly tested functionality that frustrates visitors to a Web site; security holes that jeopardize internal company systems; and other economically unpleasant or even disastrous consequences.

The general strategies for software configuration management (SCM) described in this chapter are applicable, but tactics and tools must be adapted to conform to the unique nature of WebApps. Four issues [DAR99] should be considered when developing tactics for WebApp configuration management—content, people, scalability, and politics.

Content. A typical WebApp contains a vast array of content—text, graphics, applets, scripts, audio/video files, forms, active page elements, tables, streaming data, and many others. The challenge is to organize this sea of content into a rational set of configuration objects (Section 27.1.4) and then establish appropriate configuration control mechanisms for these objects.

People. Because a significant percentage of WebApp development continues to be conducted in an ad hoc manner, any person involved in the WebApp can (and often does) create content. Many content creators have no software engineering background and are completely unaware of the need for configuration management. As a consequence, the application grows and changes in an uncontrolled fashion.

Scalability. The techniques and controls applied to a small WebApps do not scale upward well. It is not uncommon for a simple WebApp to grow significantly as interconnections with existing information systems, databases, data warehouses, and portal gateways are implemented. As size and complexity grows, small changes can have far-reaching and unintended affects that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale.

Politics. Who "owns" a WebApp? This question is argued in companies large and small, and its answer has a significant impact on the management and control activities associated with WebE. In some instances Web developers are housed outside the IT organization, creating potential communication difficulties. Dart [DAR99] suggests the following questions to help understand the politics associated with WebE:

How do I determine who has responsibility for WebApp CM?

- Who assumes responsibility for the accuracy of the information on the Web site?
- Who assures that quality control processes have been followed before information is published to the site?
- Who is responsible for making changes?
- Who assumes the cost of change?

The answers to these questions help determine the people within an organization who must adopt a configuration management process for WebApps.

27.4.2 WebApp Configuration Objects

WebApps encompass a broad range of configuration objects—content objects (e.g., text, graphics, images, video, audio), functional components (e.g., scripts, applets), and interface objects (e.g., COM or CORBA). WebApp objects can be identified (assigned file names) in any manner that is appropriate for the organization. However, the following conventions are recommended to ensure that cross-platform compatibility is maintained: filenames should be limited to 32 characters in length, mixed-case or all-caps names should be avoided, and the use of underscores in file names should be avoided. In addition, URL references (links) within a configuration object should always use relative paths (e.g., ../products/alarmsensors.html).

All WebApp content has format and structure. Internal file formats are dictated by the computing environment in which the content is stored. However, *rendering format* (often called *display format*) is defined by the aesthetic style and design rules established for the WebApp. *Content structure* defines a content architecture; that is, it defines the way in which content objects are assembled to present meaningful information to an end-user. Boiko [BOI02] defines structure as "maps that you lay over a set of content chunks [objects] to organize them and make them accessible to the people who need them."

27.4.3 Content Management

Content management is related to configuration management in the sense that a content management system (CMS) establishes a process (supported by appropriate tools) that acquires existing content (from a broad array of WebApp configuration objects), structures it in a way that enables it to be presented to an end-user, and then provides it to the client-side environment for display.

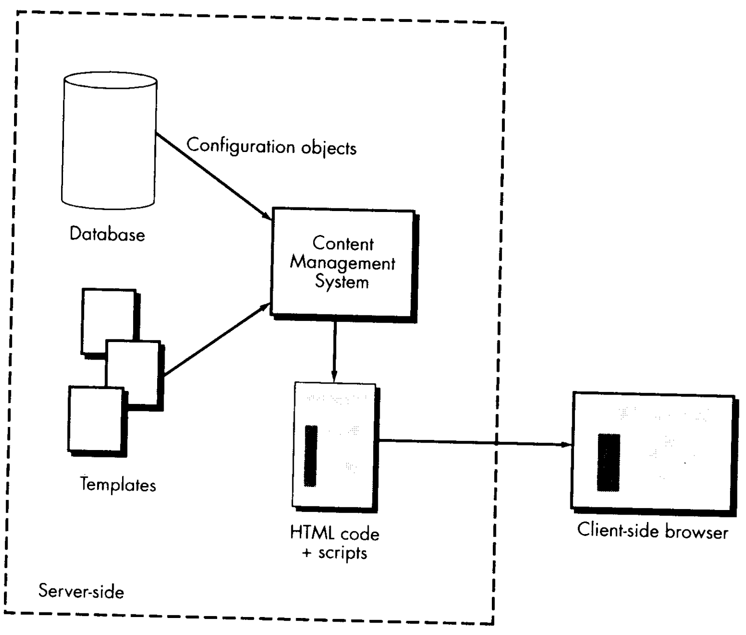
> **"Content management is an antidote to today's information frenzy."**
>
> **Bob Boiko**

The most common use of content management system occurs when a dynamic WebApp is built. Dynamic WebApps create Web pages "on-the-fly." That is, the user typically queries the WebApp requesting specific information. The WebApp queries a database, formats the information accordingly, and presents it to the user. For example, a music company provides a library of CDs for sale. When a user requests a CD or its e-music equivalent, a database is queried, and a variety of information about the artist, the CD (e.g., its cover image or graphics), the music~~al content, and~~ sample audio are all downloaded and configured into a standard content template. The resultant Web page is built on the server-side and passed to the client-side browser for examination by the end-user. A generic representation of this is shown in Figure 27.6.

KEY POINT The collection subsystem encompasses all actions required to create, acquire, and/or convert content into a form that can be presented on the client side.

In the most general sense, a CMS "configures" content for the end-user by invoking three integrated subsystems: a collection subsystem, a management subsystem, and a publishing subsystem [BOI02].

FIGURE 27.6 **Content management system (CMS)**



KEY POINT
The management subsystem implements a repository for all content. Configuration management is performed within this subsystem.

The collection subsystem. Content is derived from data and information that must be created or acquired by a content developer. The *collection subsystem* encompasses all actions required to create and/or acquire content, and the technical functions that are necessary to (1) convert content into a form that can be represented by a mark-up language (e.g., HTML, XML), and (2) organize content into packets that can be displayed effectively on the client side.

The management subsystem. Once content exists, it must be stored in a repository, cataloged for subsequent acquisition and use, and labeled to define (1) current status (e.g., is the content object complete or in development), (2) the appropriate version of the content object, and (3) related content objects. Therefore, the *management subsystem* implements a repository that encompasses the following elements:

- *Content database*—the information structure that has been established to store all content objects.
- *Database capabilities*—functions that enable the CMS to search for specific content objects (or categories of objects), store and retrieve objects, and manage the file structure that has been established for the content.
- *Configuration management functions*—the functional elements and associated workflow that support content object identification, version control, change management, change auditing, and reporting.

In addition to these elements, the management subsystem implements an administration function that encompasses the metadata and rules that control the overall structure of the content and the manner in which it is supported.

KEY POINT
The publishing subsystem extracts content from the repository and delivers it to client-side browsers.

The publishing subsystem. Content must be extracted from the repository, converted to a form that is amenable to publication, and formatted so that it can be transmitted to client-side browsers. The *publishing subsystem* accomplishes these tasks using a series of templates. Each *template* is a function that builds a publication using one of three different components [BOI02]:

- *Static elements*—text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side.
- *Publication services*—function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
- *External services*—provide access to external corporate information infrastructure such as enterprise data or "back-room" applications.

A content management system that encompasses each of these subsystems is applicable for major Web engineering projects. However, the basic philosophy and functionality associated with a CMS are applicable to all dynamic WebApps.

SOFTWARE TOOLS

Content Management

Objective: To assist software engineers and content developers in managing content that is incorporated into WebApps.

Mechanics: Tools in this category enable Web engineers and content providers to update WebApp content in a controlled manner. Most establish a simple file management system that assigns page-by-page update and editing permissions for various types of WebApp content. Some maintain a versioning system so that previous versions of content can be achieved for historical purposes.

Representative Tools[7]

Content Management Tools Suite, developed by interactivetools.com (www.interactivetools. com/), is a suite of content management tools that focus on content management for specific application domains (e.g., news articles, classified ads, real estate).

ektron-CMS300, developed by ektron (www.ektron.com), is a suite of tools that provides content management capabilities as well as Web development tools.

OmniUpdate, developed by WebsiteASP, Inc. (www.omniupdate.com), is a tool that allows authorized content providers to develop controlled updates to specified WebApp content.

Tower IDM, developed by Tower Technologies (www.towertech.com), is a document processing system and content repository for managing all forms of unstructured business information—images; forms; computer-generated reports; statements and invoices; office documents; e-mail and Web content.

Additional information on SCM and content management tools for Web engineering can be found at one or more of the following Web sites:

Web Developer's Virtual Encyclopedia (www.wdlv.com), *WebDeveloper* (www.webdeveloper.com), Developer Shed (www.devshed.com), *webknowhow.net* (www.webknowhow.net), or WebReference (www.webreference.com).

27.4.4 Change Management

The workflow associated with change control for conventional software (Section 27.3.3) is generally too ponderous for Web engineering. It is unlikely that the change request, change report, and engineering change order sequence can be achieved in an agile manner that is acceptable for most WebApp development projects. How then do we manage a continuous stream of changes requested for WebApp content and functionality?

To implement effective change management within the "code and go" philosophy that continues to dominate WebApp development, the conventional change control process must be modified. Each change should be categorized into one of four classes:

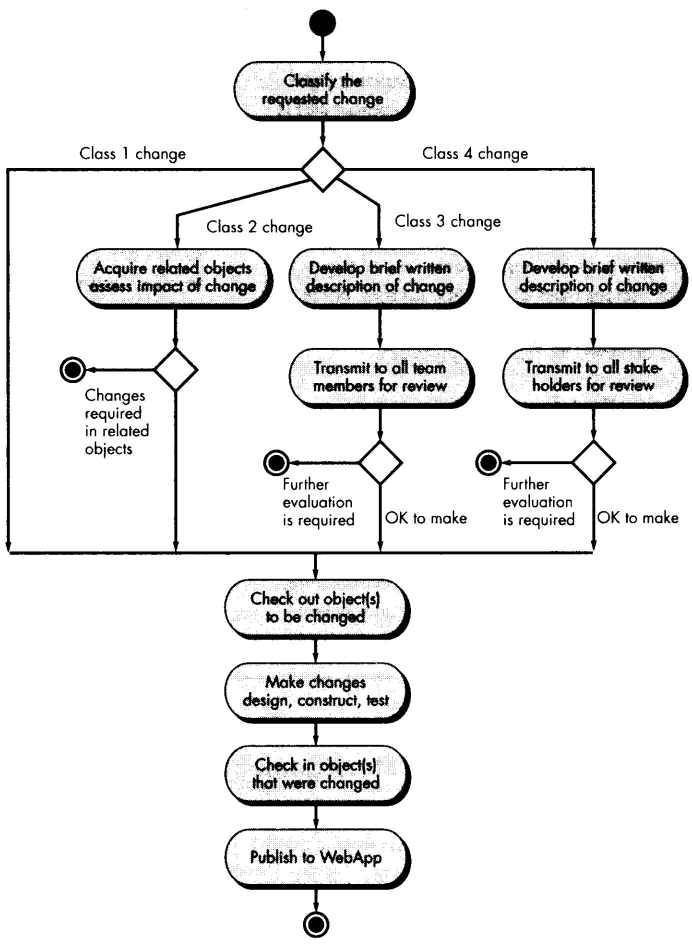
Class 1—a content or function change that corrects an error or enhances local content or functionality.

Class 2—a content or function change that has impact on other content objects or functional components.

7 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.

FIGURE 27.7

Managing changes for WebApps


Classify the requested change
Class 1 change
Class 4 change
Class 2 change
Class 3 change
Acquire related objects assess impact of change
Develop brief written description of change
Develop brief written description of change
Changes required in related objects
Transmit to all team members for review
Transmit to all stake-holders for review
Further evaluation is required
OK to make
Further evaluation is required
OK to make
Check out object(s) to be changed
Make changes design, construct, test
Check in object(s) that were changed
Publish to WebApp

Class 3—a content or function change that has broad impact across a WebApp (e.g., major extension of functionality, significant enhancement or reduction in content, major required changes in navigation).

Class 4—a major design change (e.g., a change in interface design or navigation approach) that will be immediately noticeable to one or more categories of user.

Once the requested change has been categorized, it can be processed according to the algorithm shown in Figure 27.7.

Referring to the figure, class 1 and 2 changes are treated informally and are handled in an agile manner. For a class 1 change, the Web engineer evaluates the impact of the change, but no external review or documentation is required. As the change is made, standard check-in and check-out procedures are enforced by configuration repository tools. For class 2 changes, it is incumbent on the Web engineer to review the impact of the change on related objects (or to ask other developers responsible for those objects to do so). If the change can be made without requiring significant changes to other objects, modification occurs without additional review or documentation. If substantive changes are required, further evaluation and planning are necessary.

Class 3 and 4 changes are also treated in an agile manner, but some descriptive documentation and more formal review procedures are required. A *change description*—describing the change and providing a brief assessment of the impact of the change—is developed for class 3 changes. The description is distributed to all members of the Web engineering team who review it to better assess its impact. A change description is also developed for class 4 changes, but in this case, the review is conducted by all stakeholders.

SOFTWARE TOOLS

Change Management

Objective: To assist Web engineers and content developers in managing changes as they are made to WebApp configuration objects.

Mechanics: Tools in this category were originally developed for conventional software, but can be adapted by Web engineers to make controlled changes to WebApps.

Representative Tools[8]

ChangeMan WCM, developed by Serena (www.serena.com), is a one of a suite of change management tools that provide SCM capabilities.

ClearCase, developed by Rational (www.rational.com), is a suite of tools that provides full configuration management capabilities for WebApps.

PVCS, developed by Merant (www.merant.com), is a suite of tools that provides full configuration management capabilities for WebApps.

Source Integrity, developed by mks (www.mks.com), is a SCM tool that can be integrated with selected development environments.

27.4.5 Version Control

As a WebApp evolves through a series of increments, a number of different versions may exist at the same time. One version (the current operational WebApp) is available via the Internet for end-users; another version (the next WebApp increment) may be in the final stages of testing prior to deployment; a third version is in development and represents a major update in content, interface aesthetics, and functionality. Configuration objects must be clearly defined so that each can be associated with the appropriate version. In addition, control mechanisms must be

8 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.

established. Dreilinger [DRE99] discusses the importance of version (and change) control when he writes:

In an *uncontrolled* site where multiple authors have access to edit and contribute, the potential for conflict and problems arises—more so when these authors work from different offices at different times of day and night. You may spend the day improving the file *index.html* for a customer. After you've made your changes, another developer who works at home after hours, or in another office, may spend the night uploading their own newly revised version of the file *index.html*, completely overwriting your work with no way to get it back!

This situation should sound familiar to every software engineer as well as every Web engineer. To avoid it, a version control process should be established.

1. *A central repository for the WebApp project should be established.* The repository will hold current versions of all WebApp configuration objects (content, functional components, and others).
2. *Each Web engineer creates his or her own working folder.* The folder contains those objects that are being created or changed at any given time.
3. *The clocks on all developer workstations should be synchronized.* This is done to avoid overwriting conflicts when two developers make updates that are very close to one another in time.
4. *As new configuration objects are developed or existing objects are changed, they are imported into the central repository.* The version control tool (see discussion of CVS earlier in this chapter) will manage all check-in and check-out functions from the working folders of each Web engineer.
5. *As objects are imported or exported from the repository, an automatic, time-stamped log message is made.* This provides useful information for auditing and can become part of an effective reporting scheme.

The version control tool maintains different versions of the WebApp and can revert to an older version if required.

27.4.6 Auditing and Reporting

In the interest of agility, the auditing and reporting functions are deemphasized in Web engineering work. However, they are not eliminated altogether. All objects that are checked into or out of the repository are recorded in a log that can be reviewed at any point in time. A complete log report can be created so that all members of the Web engineering team have a chronology of changes over a defined period of time. In addition, an automated e-mail notification (addressed to those developers and stakeholders who have interest) can be sent every time an object is checked in or out of the repository.

INFO

SCM Standards

The following list of SCM standards (extracted in part from www.12207.com) is reasonably comprehensive:

IEEE Standards	**standards.ieee.org/catalog/olis/**
IEEE 828	Software Configuration Management Plans
IEEE 1042	Software Configuration Management
ISO Standards	**www.iso.ch/iso/en/ISOOnline.frontpage**
ISO 10007-1995	Quality Management, Guidance for CM
ISO/IEC 12207	Information Technology—Software Life Cycle Processes
ISO/IEC TR 15271	Guide for ISO/IEC 12207
ISO/IEC TR 15846	Software Engineering—Software Life Cycle Process—Configuration Management for Software
EIA Standards	**www.eia.org/**
EIA 649	National Consensus Standard for Configuration Management
EIA CMB4-1A	Configuration Management Definitions for Digital Computer Programs
EIA CMB4-2	Configuration Identification for Digital Computer Programs
EIA CMB4-3	Computer Software Libraries
EIA CMB4-4	Configuration Change Control for Digital Computer Programs
EIA CMB6-1C	Configuration and Data Management References
EIA CMB6-3	Configuration Identification
EIA CMB6-4	Configuration Control
EIA CMB6-5	Textbook for Configuration Status Accounting
EIA CMB7-1	Electronic Interchange of Configuration Management Data
U.S. Military Standards	**www-library.itsi.disa.mil**
DoD MIL STD-973	Configuration Management
MIL-HDBK-61	Configuration Management Guidance
Other standards	
DO-178B	Guidelines for the Development of Aviation Software
ESA PSS-05-09	Guide to Software Configuration Management
AECL CE-1001-STD rev.1	Standard for Software Engineering of Safety Critical Software
DOE SCM checklist	cio.doe.gov/ITReform/sqse/download/cmcklst.doc
BS-6488	British Std., Configuration Management of Computer-Based Systems
Best Practice—UK	Office of Government Commerce: www.ogc.gov.uk
CMII	Institute of CM Best Practices: www.icmhq.com

A *Configuration Management Resource Guide* provides complementary information for those interested in CM processes and practice. It is available at www.quality.org/config/cm-guide.html.

27.5 Summary

Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies, controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of software engineering becomes part of a software configuration. The configuration is organized in a manner that enables orderly management of change.

The software configuration is composed of a set of interrelated objects, also called software configuration items, that are produced as a result of some software engineering activity. In addition to documents, programs, and data, the development environment that is used to create software can also be placed under configuration control. All SCIs are stored within a repository that implements mechanisms and data structures to ensure data integrity, provides integration support for other software tools, supports information sharing among all members of the software team, and implements functions in support of version and change control.

Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baselined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and composite objects form an object pool from which versions are created. Version control is the set of procedures and tools for managing the use of these objects.

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

Configuration management for Web engineering is similar in most respects to SCM for conventional software. However, each of the core SCM tasks should be streamlined to make it as lean as possible, and special provisions for content management must be implemented.

References

[BAB86] Babich, W.A., *Software Configuration Management,* Addison-Wesley, 1986.

[BAC98] Bach, J., "The Highs and Lows of Change Control," *Computer,* vol. 31, no. 8, August 1998, pp. 113–115.

[BER80] Bersoff, E.H., V.D. Henderson, and S.G. Siegel, *Software Configuration Management,* Prentice-Hall, 1980.

[BOI02] Boiko, B., *Content Management Bible,* Hungry Minds Publishing, 2002.

[CHO89] Choi, S.C., and W. Scacchi, "Assuring the Correctness of a Configured Software Description," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 66–75.

[CVS02] Concurrent Versions System Web site, www.cvshome.org, 2002.

[DAR91] Dart, S., "Concepts in Configuration Management Systems," *Proc. Third International Workshop on Software Configuration Management,* ACM SIGSOFT, 1991, download from: http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.

[DAR99] Dart, S., "Change Management: Containing the Web Crisis," *Proc. Software Configuration Management Symposium,* Toulouse, France, 1999, available at http://www.perforce.com/perforce/conf99/dart.html.

[DAR01] Dart, S., *Spectrum of Functionality in Configuration Management Systems,* Software Engineering Institute, 2001, available at http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html.

[DRE99] Dreilinger, S., "CVS Version Control for Web Site Projects," 1999, available at http://www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.

[FOR89] Forte, G., "Rally Round the Repository," *CASE Outlook,* December 1989, pp. 5–27.

[GRI95] Griffen, J., "Repositories: Data Dictionary Descendant Can Extend Legacy Code Investment," *Application Development Trends,* April 1995, pp. 65–71.

[GUS89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 114–117.

[HAR89] Harter, R., "Configuration Management," *HP Professional,* vol. 3, no. 6, June 1989.

[IEE94] *Software Engineering Standards,* 1994 edition, IEEE Computer Society, 1994.

[JAC02] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More," *Cutter IT Journal,* vol. 15, no. 1., January 2002, pp. 18–24.

[REI89] Reichenberger, C., "Orthogonal Version Management," *Proc. 2nd Intl. Workshop on Software Configuration Management,* ACM, Princeton, NJ, October 1989, pp. 137–140.

[SHA95] Sharon, D., and R. Bell, "Tools That Bind: Creating Integrated Environments," *IEEE Software,* March 1995, pp. 76–85.

[TAY85] Taylor, B., "A Database Approach to Configuration Management for Large Projects," *Proc. Conf. Software Maintenance—1985,* IEEE, November 1985, pp. 15–23.

Problems and Points to Ponder

27.1. Use UML aggregations or composites (Chapter 8) to describe the interrelationships among the SCIs (configuration objects) listed in Section 27.1.4.

27.2. What are the four elements that exist when an effective SCM system is implemented? Discuss each briefly.

27.3. Discuss the reasons for baselines in your own words.

27.4. Assume that you're the manager of a small project. What baselines would you define for the project, and how would you control them?

27.5. What is the difference between a SCM audit and a formal technical review? Can their functions be folded into one review? What are the pros and cons?

26.6. Research an existing SCM tool, and describe how it implements control for versions and configuration objects in general.

27.7. Design a project database (repository) system that would enable a software engineer to store, cross-reference, trace, update, and change, all important software configuration items. How would the database handle different versions of the same program? Would source code be handled differently than documentation? How will two developers be precluded from making different changes to the same SCI at the same time?

27.8. Why is the First Law of System Engineering true? Provide specific examples for each of the four fundamental reasons for change.

27.9. Develop a checklist for use during configuration audits.

27.10. Using Figure 27.5 as a guide, develop an even more detailed work breakdown for change control. Describe the role of the CCA and suggest formats for the change request, the change report, and the ECO.

27.11. Research an existing SCM tool and describe how it implements the mechanics of version control. Alternatively, read two or three of the papers on SCM and describe the different data structures and referencing mechanisms that are used for version control.

27.12. The relations <part-of> and <interrelated>represent simple relationships between configuration objects. Describe five additional relationships that might be useful in the context of a SCM repository.

27.13. What is content management? Use the Web to research the features of a content management tool and provide a brief summary.

27.14. Briefly describe the differences between SCM for conventional software and SCM for WebApps.

Further Readings and Information Sources

Lyon (*Practical CM,* Raven Publishing, 2003, available at www.configuration.org) has written a comprehensive guide for CM professionals that includes pragmatic guidelines for implementing every aspect of a configuration management system (updated yearly). Hass (*Configuration Management: Principles and Practice,* Addison-Wesley, 2002) and Leon (*A Guide to Software Configuration Management,* Artech House, 2000) provide useful surveys of the subject. White and Clemm (*Software Configuration Management Strategies and Rational ClearCase,* Addison-Wesley, 2000) present SCM within the context of one of the more popular SCM tools.

Mikkelsen and Pherigo (*Practical Software Configuration Management: The Latenight Developer's Handbook,* Allyn & Bacon, 1997) and Compton and Callahan (*Configuration Management for Software,* VanNostrand-Reinhold, 1994) provide pragmatic tutorials on important SCM practices. Ben-Menachem (*Software Configuration Management Guidebook,* McGraw-Hill, 1994), and Ayer and Patrinnostro (*Software Configuration Management,* McGraw-Hill, 1992) present good overviews for those who need further introduction to the subject. Berlack (*Software Configuration Management,* Wiley, 1992) presents a useful survey of SCM concepts, emphasizing the importance of the repository and tools in the management of change. Babich [BAB86] provides an abbreviated, yet effective treatment of pragmatic issues in software configuration management. Arnold and Bohner (*Software Change Impact Analysis,* IEEE Computer Society Press, 1996) have edited an anthology that discusses how to analyze the impact of change within complex software-based systems.

Berczuk and Appleton (*Software Configuration Management Patterns,* Addison-Wesley, 2002) present a variety of useful patterns that assist in understanding SCM and implementing effective SCM systems. Brown, et al. (*Anti-Patterns and Patterns in Software Configuration Management,* Wiley, 1999) discuss the things not to do (anti-patterns) when implementing an SCM process and then consider their remedies.

Buckley (*Implementing Configuration Management,* IEEE Computer Society Press, 1993) considers configuration management approaches for all system elements—hardware, software, and firmware—with detailed discussions of major CM activities. Rawlings (*SCM for Network Development Environments,* McGraw-Hill, 1994) considers the impact of SCM for software development in a networked environment. Bays (*Software Release Methodology,* Prentice-Hall, 1999) presents a collection of best practices for all activities that occur after changes are made to an application.

As WebApps have become more dynamic, content management has become an essential topic for Web engineers. Books by Addey and his colleagues (*Content Management Systems,* Glasshaus, 2003), Boiko [BOI02], Hackos (*Content Management for Dynamic Web Delivery,* Wiley, 2002), Nakano (*Web Content Management,* Addison-Wesley, 2001) present worthwhile treatments of the subject.

A wide variety of information sources on software configuration management is available on the Internet. An up-to-date list of World Wide Web references can be found at the SEPA Web site: **http://www.mhhe.com/pressman.**