CHAPTER

15 REVIEW TECHNIQUES

KEY CONCEPTS

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" software engineering work products, including requirements and design models, code, and testing data. Freedman and Weinberg [Fre90] discuss the need for reviews this way:

> Technical work needs reviewing for the same reason that pencils need erasers: *To err is human.* The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else. The review process is, therefore, the answer to the prayer of Robert Burns:
>
> *O wad some power the giftie give us*
>
> *to see ourselves as other see us*
>
> A review—any review—is a way of using the diversity of a group of people to:
>
> 1. Point out needed improvements in the product of a single person or team;

QUICK LOOK

What is it? You'll make mistakes as you develop software engineering work products. There's no shame in that—as long as you try hard, very hard, to find and correct the mistakes before they are delivered to end users. Technical reviews are the most effective mechanism for finding mistakes early in the software process.

Who does it? Software engineers perform technical reviews, also called peer reviews, with their colleagues.

Why is it important? If you find an error early in the process, it is less expensive to correct. In addition, errors have a way of amplifying as the process proceeds. So a relatively minor error left untreated early in the process can be amplified into a major set of errors later in the project. Finally, reviews save time by reducing the amount of rework that will be required late in the project.

What are the steps? Your approach to reviews will vary depending on the degree of formality you select. In general, six steps are employed, although not all are used for every type of review: planning, preparation, structuring the meeting, noting errors, making corrections (done outside the review), and verifying that corrections have been performed properly.

What is the work product? The output of a review is a list of issues and/or errors that have been uncovered. In addition, the technical status of the work product is also indicated.

How do I ensure that I've done it right? First, select the type of review that is appropriate for your development culture. Follow the guidelines that lead to successful reviews. If the reviews that you conduct lead to higher-quality software, you've done it right.

Reviews are like a filter 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. Remove ineffective reviews from the flow to accelerate the process.

2. Confirm those parts of a product in which improvement is either not desired or not needed;
3. Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable.

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

15.1 Cost Impact of Software Defects

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

INFO

Bugs, Errors, and Defects

The goal of software quality control, and in a broader sense, quality management in general, 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 I make 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[1]). I 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

1 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.

clear distinction between pre- and post-release (e.g., consider an incremental process used in agile development).

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**.

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 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 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 indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, 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.

15.2 Defect Amplification and Removal

"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

A *defect amplification model* [IBM81] can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process. The model is illustrated schematically in Figure 15.1. A box represents a software engineering action. During the action, 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.

Figure 15.2 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 errors (defects) are released to the field. Figure 15.3 considers the same conditions except that design and code reviews are conducted as part of each software engineering action. In this case, 10 initial preliminary (architectural) design errors are amplified to 24 errors before testing commences. Only three latent errors exist. 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 15.2 and 15.3 is multiplied by the cost to remove an

FIGURE 15.1

Defect amplification model

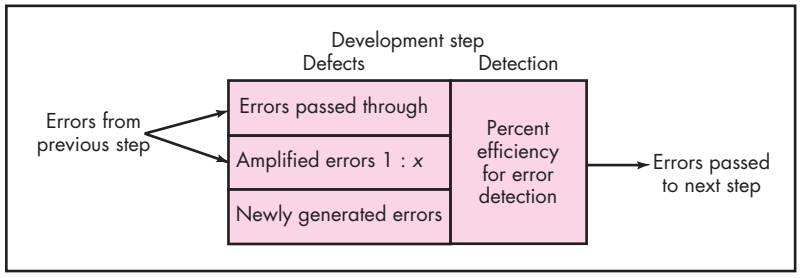


FIGURE 15.2

Defect amplification—no reviews

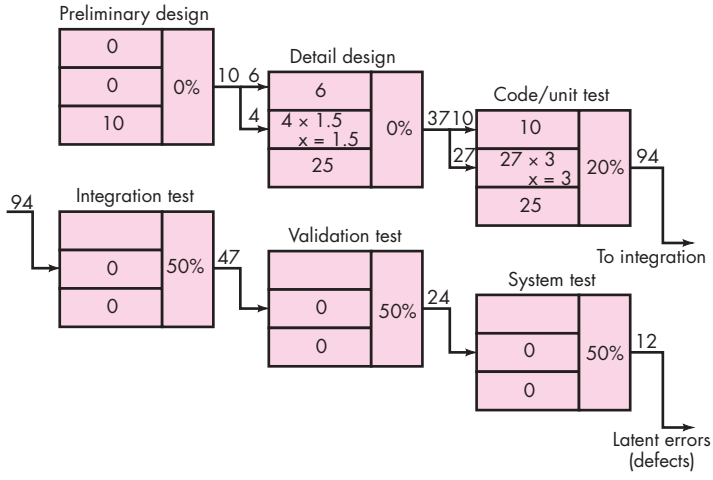
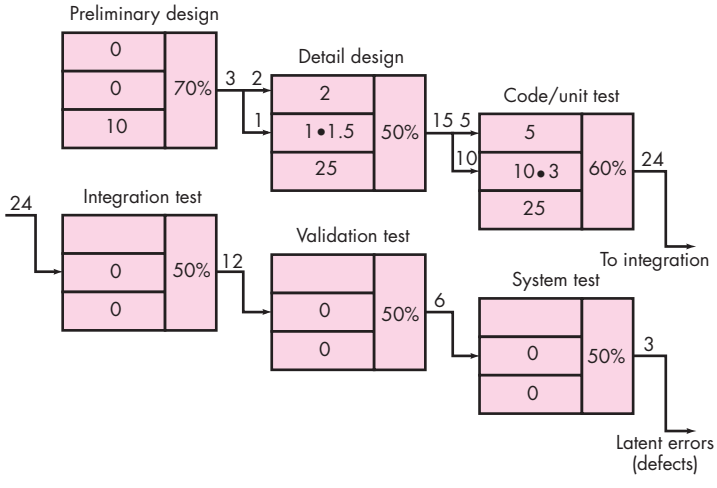


FIGURE 15.3

Defect amplification—reviews conducted



error (1.5 cost units for design, 6.5 cost units before test, 15 cost units during test, and 67 cost units after release).[2] 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, you must expend time and effort, and your development organization must spend money. However, the results of the preceding example leave little doubt that you can pay now or pay much more later.

15.3 Review Metrics and Their Use

Technical reviews are one of many actions that are required as part of good software engineering practice. Each action requires dedicated human effort, Since available project effort is finite, it is important for a software engineering organization to understand the effectiveness of each action by defining a set of metrics (Chapter 23) that can be used to assess their efficacy.

Although many metrics can be defined for technical reviews, a relatively small subset can provide useful insight. The following review metrics can be collected for each review that is conducted:

- *Preparation effort,* E_p—the effort (in person-hours) required to review a work product prior to the actual review meeting
- *Assessment effort,* E_a—the effort (in person-hours) that is expended during the actual review
- *Rework effort,* E_r—the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- *Work product size, WPS*—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- *Minor errors found,* Err_{minor}—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- *Major errors found,* Err_{major}—the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

These metrics can be further refined by associating the type of work product that was reviewed for the metrics collected.

15.3.1 Analyzing Metrics

Before analysis can begin, a few simple computations must occur. The total review effort and the total number of errors discovered are defined as:

$$E_{review} = E_p + E_a + E_r$$
$$Err_{tot} = Err_{minor} + Err_{major}$$

2 These multipliers are somewhat different than the data presented in Figure 14.2, which is more current. However, they serve to illustrate the costs of defect amplification nicely.

Error density represents the errors found per unit of work product reviewed.

$$\text{Error density} = \frac{\text{Err}_{\text{tot}}}{\text{WPS}}$$

For example, if a requirements model is reviewed to uncover errors, inconsistencies, and omissions, it would be possible to compute the error density in a number of different ways. The requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials. The review uncovers 18 minor errors and 4 major errors. Therefore, $\text{Err}_{\text{tot}} = 22$. Error density is 1.2 errors per UML diagram or 0.68 errors per requirements model page.

If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases), the percentage of errors uncovered for each review can be computed against the total number of errors found for all reviews. In addition, the error density for each work product can be computed.

Once data are collected for many reviews conducted across many projects, average values for error density enable you to estimate the number of errors to be found in a new (as yet unreviewed document). For example, if the average error density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long, a rough estimate suggests that your software team will find about 19 or 20 errors during the review of the document. If you find only 6 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.

Once testing has been conducted (Chapters 17 through 20), it is possible to collect additional error data, including the effort required to find and correct errors uncovered during testing and the error density of the software. The costs associated with finding and correcting an error during testing can be compared to those for reviews. This is discussed in Section 15.3.2.

15.3.2 Cost Effectiveness of Reviews

It is difficult to measure the cost effectiveness of any technical review in real time. A software engineering organization can assess the effectiveness of reviews and their cost benefit only after reviews have been completed, review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured (via testing).

Returning to the example presented in Section 15.3.1, the average error density for requirements models was determined to be 0.6 errors per page. The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours. The effort required for a major requirement error was found to be 18 person-hours. Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.

Requirements-related errors uncovered during testing require an average of 45 person-hours to find and correct (no data are available on the relative severity of the error). Using the averages noted, we get:

$$\text{Effort saved per error} = E_{\text{testing}} - E_{\text{reviews}}$$
$$45 - 6 = 30 \text{ person-hours/error}$$

Since 22 errors were found during the review of the requirements model, a saving of about 660 person-hours of testing effort would be achieved. And that's just for requirements-related errors. Errors associated with design and code would add to the overall benefit. The bottom line—effort saved leads to shorter delivery cycles and improved time to market.

In his book on peer reviews, Karl Wiegers [Wie02] discusses anecdotal data from major companies that have used *inspections* (a relatively formal type of technical review) as part of their software quality control activities. Hewlett Packard reported a 10 to 1 return on investment for inspections and noted that actual product delivery accelerated by an average of 1.8 calendar months. AT&T indicated that inspections reduced the overall cost of software errors by a factor of 10 and that quality improved by an order of magnitude and productivity increased by 14 percent. Others report similar benefits. Technical reviews (for design and other technical activities) provide a demonstrable cost benefit and actually save time.

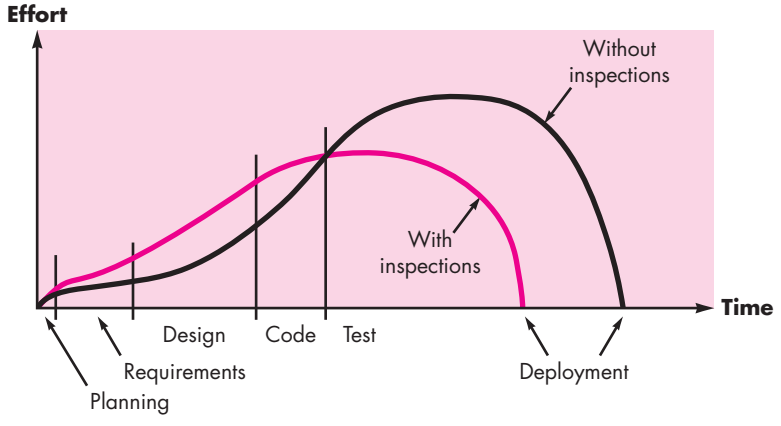
But for many software people, this statement is counterintuitive. "Reviews take time," software people argue, "and we don't have the time to spare!" They argue that time is a precious commodity on every software project and the ability to review "every work product in detail" absorbs too much time.

The examples presented earlier in this section indicate otherwise. More importantly, industry data for software reviews has been collected for more than two decades and is summarized qualitatively using the graphs illustrated in Figure 15.4.

FIGURE 15.4

Effort expended with and without reviews

Source: Adapted from [Fag86].



Referring to the figure, the effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced. As important, the deployment date for development with reviews is sooner than the deployment date without reviews. Reviews don't take time, they save it!

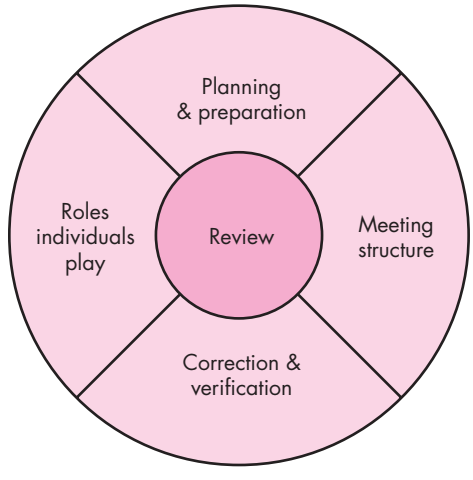
15.4 Reviews: A Formality Spectrum

Technical reviews should be applied with a level of formality that is appropriate for the product to be built, the project time line, and the people who are doing the work. Figure 15.5 depicts a reference model for technical reviews [Lai02] that identifies four characteristics that contribute to the formality with which a review is conducted.

Each of the reference model characteristics helps to define the level of review formality. The formality of a review increases when (1) distinct roles are explicitly defined for the reviewers, (2) there is a sufficient amount of planning and preparation for the review, (3) a distinct structure for the review (including tasks and internal work products) is defined, and (4) follow-up by the reviewers occurs for any corrections that are made.

To understand the reference model, let's assume that you've decided to review the interface design for **SafeHomeAssured.com**. You can do this in a variety of different ways that range from relatively casual to extremely rigorous. If you decide that the casual approach is most appropriate, you ask a few colleagues (peers) to examine the interface prototype in an effort to uncover potential problems. All of you decide that there will be no advance preparation, but that you will evaluate the prototype in a reasonably structured way—looking at layout first, aesthetics next, navigation options after that, and so on. As the designer, you decide to take a few notes, but nothing formal.

Figure 15.5
Reference model for technical reviews



But what if the interface is pivotal to the success of the entire project? What if human lives depended on an interface that was ergonomically sound? You might decide that a more rigorous approach was necessary. A review team would be formed. Each person on the team would have a specific role to play—leading the team, recording findings, presenting the material, and so on. Each reviewer would be given access to the work product (in this case, the interface prototype) before the review and would spend time looking for errors, inconsistencies, and omissions. A set of specific tasks would be conducted based on an agenda that was developed before the review occurred. The results of the review would be formally recorded, and the team would decide on the status of the work product based on the outcome of the review. Members of the review team might also verify that the corrections made were done properly.

In this book I consider two broad categories of technical reviews: informal reviews and more formal technical reviews. Within each broad category, a number of different approaches can be chosen. These are presented in the sections that follow.

15.5 Informal Reviews

Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product, or the review-oriented aspects of pair programming (Chapter 3).

A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

One way to improve the efficacy of a desk check review is to develop a set of simple review checklists for each major work product produced by the software team. The questions posed within the checklist are generic, but they will serve to guide the reviewers as they check the work product. For example, let's reexamine a desk check of the interface prototype for **SafeHomeAssured.com**. Rather than simply playing with the prototype at the designer's workstation, the designer and a colleague examine the prototype using a checklist for interfaces:

- Is the layout designed using standard conventions? Left to right? Top to bottom?
- Does the presentation need to be scrolled?
- Are color and placement, typeface, and size used effectively?
- Are all navigation options or functions represented at the same level of abstraction?
- Are all navigation choices clearly labeled?

and so on. Any errors or issues noted by the reviewers are recorded by the designer for resolution at a later time. Desk checks may be scheduled in an ad hoc manner, or they may be mandated as part of good software engineering practice. In general, the amount of material to be reviewed is relatively small and the overall time spent on a desk check spans little more than one or two hours.

In Chapter 3, I described *pair programming* in the following manner: "XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving (two heads are often better than one) and real-time quality assurance."

Pair programming can be characterized as a continuous desk check. Rather than scheduling a review at some point in time, pair programming encourages continuous review as a work product (design or code) is created. The benefit is immediate discovery of errors and better work product quality as a consequence.

In their discussion of the efficacy of pair programming, Williams and Kessler [Wil00] state:

> Anecdotal and initial statistical evidence indicates that pair programming is a powerful technique for productively generating high quality software products. The pair works and shares ideas together to tackle the complexities of software development. They continuously perform inspections on each other's artifacts leading to the earliest, most efficient form of defect removal possible. In addition, they keep each other intently focused on the task at hand.

Some software engineers argue that the inherent redundancy built into pair programming is wasteful of resources. After all, why assign two people to a job that one person can accomplish? The answer to this question can be found in Section 15.3.2. If the quality of the work product produced as a consequence of pair programming is significantly better than the work of an individual, the quality-related savings can more than justify the "redundancy" implied by pair programming.

INFO

Review Checklists

Even when reviews are well organized and properly conducted, it's not a bad idea to provide reviewers with a "crib sheet." That is, it's worthwhile to have a checklist that provides each reviewer with the questions that should be asked about the specific work product that is undergoing review.

One of the most comprehensive collections of review checklists has been developed by NASA at the Goddard Space Flight Center and is available at **http://sw-assurance.gsfc.nasa.gov/disciplines/quality/index.php**

Other useful technical review checklists have also been proposed by:

Process Impact (**www.processimpact.com/pr_goodies.shtml**)

Software Dioxide (**www.softwaredioxide.com/Channels/ConView.asp?id=6309**)

Macadamian (**www.macadamian.com**)

The Open Group Architecture Review Checklist (**www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm**)

DFAS [downloadable] (**www.dfas.mil/technology/pal/ssps/docstds/spm036.doc**)

15.6 Formal Technical Reviews

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

Quote:

"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* and *inspections.* Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs, see [Rad02], [Wie02], or [Fre90].

15.6.1 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.

WebRef

The NASA SATC *Formal Inspection Guidebook* can be downloaded from **satc.gsfc.nasa.gov/Documents/fi/gdb/fi.pdf**.

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

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

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

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

15.6.2 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.

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

15.6.3 Review Guidelines

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

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

1. *Review the product, not the producer.* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of

accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.

Quote:

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

Author unknown

2. *Set an agenda and maintain it.* One of the key maladies of meetings of all types is drift. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.
3. *Limit debate and rebuttal.* When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.
4. *Enunciate problem areas, but don't attempt to solve every problem noted.* A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. *Take written notes.* It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded. Alternatively, notes may be entered directly into a notebook computer.
6. *Limit the number of participants and insist upon advance preparation.* Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).
7. *Develop a checklist for each product that is likely to be reviewed.* A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even testing work products.

Quote:

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

Ralph Waldo Emerson

8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as tasks during the software process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.
9. *Conduct meaningful training for all reviewers.* To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg [Fre90] estimate a one-month learning curve for every 20 people who are to participate effectively in reviews.

10. *Review your early reviews.* Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

Because many variables (e.g., number of participants, type of work products, timing and length, specific review approach) have an impact on a successful review, a software organization should experiment to determine what approach works best in a local context.

15.6.4 Sample-Driven Reviews

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

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

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

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 be representative of the work product as a whole and large enough to be meaningful to the reviewers who do 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.[3]

3 Thelin and his colleagues have conducted a detailed simulation that can assist in making this determination. See [The01] for details.

SafeHome

Quality 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 other members of the product software engineering team.

The conversation:

Doug: I know we didn't spend time developing a quality plan for this project, but we're already into it and we have to consider quality … right?

Jamie: Sure. We've already decided that as we develop the requirements model [Chapters 6 and 7], Ed has committed to develop a testing 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.

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 requirements 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 15.6.4] 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 3]. 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.

15.7 Summary

The intent of every technical review is to find errors and uncover issues that would have a negative impact on the software to be deployed. The sooner an error is uncovered and corrected, the less likely that error will propagate to other software engineering work products and amplify itself, resulting in significantly more effort to correct it.

To determine whether quality control activities are working, a set of metrics should be collected. Review metrics focus on the effort required to conduct the review and the types and severity of errors uncovered during the review. Once metrics data are collected, they can be used to assess the efficacy of the reviews you do conduct. Industry data indicates that reviews provide a significant return on investment.

A reference model for review formality identifies the roles people play, planning and preparation, meeting structure, correction approach, and verification as the characteristics that indicate the degree of formality with which a review is conducted. Informal reviews are casual in nature, but can still be used effectively to uncover errors. Formal reviews are more structured and have the highest probability of leading to high-quality software.

Informal reviews are characterized by minimal planning and preparation and little record keeping. Desk checks and pair programming fall into the informal review category.

A formal technical review is a stylized meeting that has been shown to be extremely effective in uncovering errors. Walkthroughs and inspections establish defined roles for each reviewer, encourage planning and advance preparation, require the application of defined review guidelines, and mandate record keeping and status reporting. Sample-driven reviews can be used when it is not possible to conduct formal technical reviews for all work products.

Problems and Points to Ponder

15.1. Explain the difference between an *error* and a *defect.*

15.2. Why can't we just wait until testing to find and correct all software errors?

15.3. Assume that 10 errors have been introduced in the requirements model and that each error will be amplified by a factor of 2:1 into design and an addition 20 design errors are introduced and then amplified 1.5:1 into code where an additional 30 errors are introduced. Assume further that all unit testing will find 30 percent of all errors, integration will find 30 percent of the remaining errors, and validation tests will find 50 percent of the remaining errors. No reviews are conducted. How many errors will be released to the field.

15.4. Reconsider the situation described in Problem 15.3, but now assume that requirements, design, and code reviews are conducted and are 60 percent effective in uncovering all errors at that step. How many errors will be released to the field?

15.5. Reconsider the situation described in Problems 15.3 and 15.4. If each of the errors released to the field costs $4,800 to find and correct and each error found in review costs $240 to find and correct, how much money is saved by conducting reviews?

15.6. Describe the meaning of Figure 15.4 in your own words.

15.7. Which of the reference model characteristics do you think has the strongest bearing on review formality? Explain why.

15.8. Can you think of a few instances in which a desk check might create problems rather than provide benefits?

15.9. 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?

15.10. Considering all of the review guidelines presented in Section 15.6.3, which do you think is most important and why?

Further Readings and Information Sources

There have been relatively few books written on software reviews. Recent editions that provide worthwhile guidance include books by Wong (*Modern Software Review,* IRM Press, 2006), Radice (*High Quality, Low Cost Software Inspections,* Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001), and Gilb and Graham (*Software Inspection,* Addison-Wesley, 1993). Freedman and Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews,* Dorset House, 1990) remains a classic text and continues to provide worthwhile information about this important subject.

A wide variety of information sources on software reviews is available on the Internet. An up-to-date list of World Wide Web references relevant to software reviews can be found at the SEPA website: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.