CHAPTER FIVE

Reviews

A **REVIEW IS ANY ACTIVITY IN WHICH A WORK PRODUCT** is distributed to reviewers who examine it and give feedback. Different work products will go through different kinds of reviews: the team may do a very thorough, technical review of a software requirements specification, while the vision and scope document will be passed around via email and have higher-level walkthroughs. (This book was reviewed by a wide range of experts including seasoned project managers, university faculty, and business executives.) This chapter covers several kinds of reviews, each of which may be appropriate for different work products and at various points during the software project.

Reviews are useful not only for finding and eliminating defects, but also for gaining consensus among the project team, securing approval from stakeholders, and aiding in professional development for team members. In all cases, the work product coming out of the review has fewer defects than it had when it was submitted—even though the author thought it was "complete" before the review. Every defect that is found during a review is a defect that someone did not have to spend time tracking down later in the project.

There are many ways that a work product can be reviewed. Each kind of review is appropriate for different audiences or kinds of work product. The purpose of all reviews is to ensure that each reviewer is satisfied that the work product is correct, and that his or her perspective is represented.

The goal of every review is to save the project team time and effort. It's much easier to fix the problems on paper, before they cause software to be built incorrectly. An effective way to make sure defects are caught early is to schedule many reviews over the course of the project to catch the defects before they become deeply embedded in the software. By reviewing each work product before it is approved, a project manager sets those checkpoints and ensures that defects are caught early, before they are repeated in later work products.

Inspections

An *inspection* is one of the most common sorts of review found in software projects. The goal of the inspection is for all of the inspectors to reach consensus on a work product and approve it for use in the project. Commonly inspected work products include software requirements specifications (see Chapter 6) and test plans (see Chapter 8). In an inspection, a work product is selected for review and a team is gathered for an inspection meeting to review the work product. A moderator is chosen to moderate the meeting. Each inspector prepares for the meeting by reading the work product and noting each defect. In an inspection, a defect is any part of the work product that will keep an inspector from approving it. For example, if the team is inspecting a software requirements specification, each defect will be text in the document that an inspector disagrees with. The goal of the inspection is to repair all of the defects so that everyone on the inspection team can approve the work product.

Most project managers have seen their projects get delayed because of scope creep and unnecessary work caused by changes that, had they been caught earlier, would have required much less work to fix. One of the most common complaints from project team

members is that they would have built the software differently had they been given a better understanding of what was needed from the beginning. One important root cause of these problems is defects in work products that are not caught until long after those work products have been used as the basis for later project activities.

The most important reason to inspect documents is to prevent defects. If the team starts building software based on a vision and scope document that has a serious defect, eventually the entire project will have to stop and reverse course. This can be very expensive in both effort and morale, because the team will need to backtrack and revise the requirements, design, code, test plans, and other work products that they have already put a lot of effort into implementing. The same goes for defects that were caught in other work products—defects missed in a design specification, for example, will have to be corrected later after they have been coded. The longer a defect goes uncorrected, the more entrenched it is in other work products; the more entrenched it is, the more time and effort it will take to fix.

According to a report by the Software Engineering Institute, it costs more to not do inspections than it does to do them. A national survey of software engineering teams found that in a typical inspection, four to five people spend one to two hours preparing for inspections, followed by one to two hours to hold an inspection meeting. The total effort required for the inspection, therefore, is 10 to 20 person-hours; this effort results in the early detection of an average of 5 to 10 defects. (On the average, these defects, if left in the document, would require either 250 to 500 lines of new code or modification of 1000 to 1500 lines of legacy code to repair if they were eventually caught—which would almost certainly require well over 20 person-hours of programmers' time!) This is a very high return on investment; few tools, techniques, or practices are as effective as inspections for increasing the quality of the software.

Inspections are easy to implement, and have an immediate effect on quality and consensus-building. A small team spending a few hours inspecting a work product will catch errors that could potentially save weeks, or even months, of wasted effort. An effective inspection requires a well-chosen team, a moderator who is able to run the meeting, and an author who is willing to listen to criticism and fix the work product being inspected.

Choose the Inspection Team

The job of the inspection team is to work with the author of the document in order to identify any defects. A *defect* is any problem in the document that will prevent an inspector from approving it. Once a problem is identified, the inspection team must work to come up with a solution that will fix the problem. When the team meets to inspect the document, they will be expected to come up with solutions to the defects that they find. The project manager must select a team that can perform this function. This means that each inspector needs to have enough familiarity with the project and the way the work product will be used to understand its problems and propose changes. (Team members who use the document in their day-to-day work should have no problem with this.)

The project manager must choose a team of 3 to 10 inspectors. Ideally, each inspector should represent a different perspective on the work product. A designer will use a document for different tasks than a programmer will. It is important that each person who will use the document has his views represented in the inspection team. This is critical for catching all of the defects.

During the inspection, the team works to identify any defects in the work product. They are expected to evaluate it from two perspectives. The most important evaluation is from the perspective of their own expertise, where the inspectors identify any issues that will interfere with the development of the project. For this role, they must draw on their engineering skills and experience with past software projects. But inspectors should also evaluate the work product from a common sense perspective. Each inspection team member should think about the ideas in the vision and scope document and consider several questions: does the work product being inspected fulfill the needs laid out in the vision and scope document? Does it really answer the problem posed by earlier work products? Will it be able to serve as a basis for all of the work products that will come later in the product? Good inspection team members will be able to keep these questions in mind.

Select a Moderator

The project manager must choose a moderator to run the inspection meetings. This person must be able to objectively evaluate the work product being inspected and understand any issues that are raised during the inspection. The moderator will also need to be able to control the meeting. If a few inspectors start a discussion to address a defect that might take a lot of time, the moderator will have to be able to stop that discussion and table it as an open issue. It takes some practice to keep control of a meeting.

The project manager should be an inspector, so an independent and unbiased moderator is needed. A good moderator will have sufficient technical background to understand the work product being inspected. It is important for the moderator to be objective, and not to favor one perspective over another during the inspection meeting. In some organizations, the moderator is never a part of the project team, and does not have a stake in the project. However, some people have found it useful to select as moderator a team member who will not be inspecting the document, because the moderator should have an understanding of the issues discussed during the meeting. But in that case, that team member must be willing and able to stay objective by allowing every inspector equal opportunity to bring up defects, and by ensuring that each issue is discussed and either resolved or tagged as an open issue.

The hardest part of the moderator's job is to prepare the inspectors and the author for criticism of the work product. When somebody writes a document, he may be uncomfortable with the idea that it contains errors. It's his work and, in our day-to-day lives, few of us are used to having our work critiqued. But all documents have defects, and authors need to get comfortable with this idea. This is by far the most challenging part of implementing inspections: getting people comfortable with having their work criticized.

To address this, the moderator must help the author understand the benefit of the criticism. It's the moderator's job to make sure that the meeting does not become personal criticism, and that the comments are always constructive. An effective way to do this is to focus the discussion on each defect and come up with a specific resolution. It's the job of the inspection team to do more than just identify the problems; they must also come up with the solutions. The moderator compiles all of the defect resolutions into an *inspection log* (see Figure 5-1 for an example).

# of issues:	16		
Review date:	March 16, 2003		
Attendees		**Read document**	**Time spent preparing**
Mike (project manager)		Y	Author
Barbara (VP)		Y	1.0 hours
Quentin (requirements analyst)		Y	2.0 hours
Sophie (senior QA engineer		Y	3.0 hours
Jill (senior programmer)		Y	0.5 hours
Issue no.	**Section/page**	**Identified by**	**Issue**
1	Global	Quentin	The term "standard contract" should be replaced with "pro-forma contract."
2	Section 3.1.1 Line 165	Sophie	The contents of the cells in the table are out of order. It looks like some cells were marked down.
3	Section 3.1.2 Line 190	Jill	Specify the look up is by contract number and artist name.
4	Section 3.3b Line 623	Sophie	Tile of the section needs to be changed to "Deletion file (maintenance)." To be consistent with section 3.2.1 #1.

FIGURE 5-1. Sample inspection log

At the top of each inspection log is information about the inspection meeting: what work product was being reviewed, when it was held, who was in attendance, whether or not the work product was read by each inspector, how long each inspector spent reviewing the work product, and how many issues (including both defects and open issues) were found. Each work product should have a unique version number, to ensure that the inspection log can be matched up to the proper version of the work product.

The inspection moderator should ask each team member how long he or she spent reviewing the work product and record that number in the log. This stands as a record of how much effort went into the work product, which will help in future estimation, project planning, and impact analysis activities. If any inspector failed to review the work product, the moderator must halt the meeting and reschedule it in order to allow all of the inspectors enough time to review the work product.

The rest of the inspection log contains a list of action items. Each item is marked with the exact location of the defect and the solution proposed by the inspection team. In many cases, the solution is a wording change. The work product contains a specific sentence that is unclear, ambiguous, or incorrect. After a brief (usually under five minutes) discussion, the inspection team identifies wording that will correct the defect. In this case, the moderator writes the new wording in the inspection log, and the author agrees to update the work product accordingly.

In other cases, the solution is too complex to be identified at the inspection meeting. If the discussion lasts too long, the moderator should stop it and add an open issue to the inspection log. This issue must be assigned to the author and, optionally, one or more inspection team members; it is their responsibility to resolve the issue. The log item for this must contain a specific set of actions to be performed ("meet with Marketing to research this missing feature," "rewrite lines 534 through 539 so system will perform data check on report B").

It is important that the inspection log is readily available to all inspectors. After the meeting, it should be distributed to all inspectors, and stored along with previous versions of the document. If the document is checked into a version control system (see Chapter 7), then each inspection log should be checked in along with it. All changes must be made before the work product can be inspected again: open issues must be closed, defects must be repaired, and every issue reported in the log must be addressed.

Inspect the Work Product

During the inspection meeting, a moderator leads the team page by page through a printed copy of the work product. The purpose of the meeting is to identify and fix any defects. The moderator does not actually read each page out loud or give the team time to read the page. The team members read the document prior to the inspection, during their preparation. When the moderator goes through the document page by page, he simply asks the reviewers for their defects on page 1; once those are done, he asks for the defects on page 2 and continues through the rest of the document.

Prior to the inspection meeting, each team member should be given a checklist to help her identify defects. Checklists will be different for different kinds of work products. (In other chapters, checklists will be included for each type of work product that should be inspected.) The script in Table 5-1 describes the process for an inspection meeting.

TABLE 5-1. Inspection meeting script

Name	**Inspection meeting script**
Purpose	To run a moderated inspection meeting
Summary	In an inspection meeting, a moderator leads a team of reviewers in reviewing a work product and fixing any defects that are found.
Work Products	*Input* Work product being inspected *Output* Inspection log

TABLE 5-1. Inspection meeting script (continued)

Name	Inspection meeting script
Entry Criteria	A moderator must be selected, as well as team of 3 to 10 people. A work product must be selected, and each team member has read it individually and identified all wording that must be changed or clarified before he or she will approve the work product. A unique version number has been assigned to the work product.
Basic Course of Events	1. *Preparation.* The moderator distributes a printed version of the work product (with line numbers) to each inspector, along with a checklist to aid in the review. Each inspector reads the work product and identifies any defects to be brought up at the meeting. 2. *Overview.* The inspection meeting begins. The moderator verifies that each team member is prepared. 3. *Page-by-page review.* The moderator runs through the work product page by page. Inspectors indicate where there are defects. Each defect is either resolved or left as an open issue. The moderator adds each defect to the inspection log. 4. *Rework.* The author repairs the defects identified in the inspection meeting. 5. *Follow-up.* Inspection team members verify that the defects were repaired. 6. *Approval.* The inspection team approves the work product.
Alternative Paths	1. During Step 2, if any team member has not read the work product, then the inspection is halted. The meeting is rescheduled and the script returns to step 1. 2. During Step 4, if an inspection team member discovers additional defects in the work product, then the moderator calls another meeting and the process returns to step 1.
Exit Criteria	The work product has been approved.

Preparation

Each inspector reviews the printed copy of the work product individually, prior to the inspection meeting. Any defects that are found should be marked on the copy so that they can be brought up in the meeting.

In many organizations, the moderator requires that each inspector submit a written list of defects that were found prior to the inspection meeting, and all defects are compiled into a single inspection log and distributed to the entire inspection team. This optional step can reduce the time required for the meeting because instead of going through the entire work product page by page, the moderator only goes through the log, and the author and inspectors have time to prepare in advance to respond to the defects.

Overview

The moderator verifies that each inspection team member has read the printed copy of the work product. If any team member has not prepared, the inspection is aborted and rescheduled for a later date.

Page-by-page review

The moderator turns to the first page of the work product and asks if anyone found any issues on that page. Team members bring up each issue that they found during their preparation. For each issue, the moderator leads a discussion between the team and the author to identify new wording that will resolve the issue. (For work products that are not text or documents, the team describes the change in sufficient detail so that the repair of the defect is unambiguous to the author.) The team should come up with the actual text that will be inserted into the document in order to fix the defect; the moderator should add this fix to the inspection log. If the team cannot come up with a fix on

the spot, or if discussion lasts more than about five minutes, the moderator adds it to the inspection log as an open issue and assigns it to the team member who brought it up (and anyone else who is involved), so he can work with the author to resolve it. Once all issues for the page are discussed, the moderator moves to the next page in the work product.

Rework

After the inspection meeting is over, the author makes the changes in the inspection log and works with the inspection team members to resolve all open issues. When the changes are complete, the author turns the updated work product over to the moderator.

Follow-up

The moderator distributes the updated work product to the inspection team. Each team member verifies that he can now approve the work product. If there are any issues that were not fully resolved or additional defects that were not caught, he notifies the moderator, who calls another inspection meeting and starts the inspection process over again. Once the team gets through an inspection without any open issues and can agree on any changes that must be made, the work product can be updated and distributed for approval.

Approval

If any inspector feels that there are still further issues raised by the corrections to the work product, another inspection meeting can be held; however, the project manager and author can also work individually with everyone involved to make sure that the changes are adequate. Once everyone on the team feels that the changes they identified are adequate, they can approve the updated work product without holding another inspection meeting.

The moderator adds a signature page to the work product and distributes a printed version for signature approval. The signed work product is archived.

Manage the Author's Expectations

Many people who have implemented inspections have found that it is very difficult for authors to sit through an inspection meeting without defending their work. Instead of providing clarification that is used to update the work product, they take over the discussion and, by being defensive and loud, get the inspectors to agree not to report defects. This is counterproductive: it leads to situations when the inspection team understands what the author meant, but the work product, which remains unchanged, does not reflect this. It is the moderator's job to keep the discussions on track and prepare the authors for the inspection.

A major challenge of the moderator role is keeping the author from altering the understanding of the document through discussion. The purpose of the discussions is not to teach the inspectors about the project; it is to change the document so that the author and all of the inspectors will approve it. There is a simple rule in document inspection: if there is a misunderstanding about words in the document, they need to be clarified in the document, and not just in the minds of the people who happened to attend the inspection meeting.

Each inspector should keep in mind the fact that if he did not understand something after reading a document, then it is probably the document's fault, not the reader's. If he did not understand it, then it is likely that another reader will also have the same problem (especially considering that most software documents will be used as reference later, by people who are less familiar with the project than the inspectors). For this reason, it is very important that inspectors make it clear when they do not understand something. This is difficult for many inspectors: it's hard for people to admit that they did not understand something that they have read. It is the moderator's job to draw these misunderstandings out of the inspectors during the discussions of each defect.

The author should be prepared to listen to the inspection team discuss defects. It is tempting to get defensive and try to defend each defect. The author must remember that if someone thinks that an issue is worth bringing up in the meeting, there may be some ambiguity there, no matter how clear it seems to the person who wrote the words.

One way to help the author feel less defensive is to take the option (described above, under "Preparation") in which the inspection team members submit their defects to the moderator before the inspection meeting. The moderator compiles all of the defects into a log, which is then sent back to the inspection team. This is helpful because it gives the author advance warning of all of the defects that will be discussed. It also allows the inspection team to prepare solutions to the defects in advance. However, it requires more effort on the part of the moderator, who has to look through all of the defects up front in order to group redundant defects together and make sure that each one is described clearly.

In some organizations, project managers have found it useful to require that the authors not talk in the inspection meetings, to let the document stand on its own. In others, the author is excluded from the meeting entirely, and is simply given the inspection log. Although this sounds drastic and impersonal to some people, some moderators have found this to be a very useful practice, as the team feels that they must put a lot of effort into making the inspection log as self-explanatory as possible. However, while these practices do prevent the author from skewing the results of the inspection, they also cause the author to miss out on important discussions; this is a costly trade-off. As long as the author is able to listen to the moderator's rules, especially when it comes to identifying and addressing defects, he can be a valuable participant in the inspection process.

Help Others in the Organization Accept Inspections

Over the many years that inspections have been practiced in software organizations, project managers have often found that when they attempt to implement inspections, the team pushes back. This opposition occurs because, to some people, it is not intuitively obvious that spending the time inspecting the work products up front will save the team from having to fix the software later. The project manager should prepare for potential resistance by understanding exactly why inspections are important.

Project managers often find that engineers are very unhappy with the idea of inspections. To some, inspections seem unnecessarily "bureaucratic." This is especially unfortunate

because inspections are one of the most effective ways to prevent defects and make the most efficient use of the engineers' time. There are few tools or techniques that have such a high potential savings in effort. For each hour spent inspecting documents, the team saves many hours that would otherwise be lost on correcting problems that would have been coded incorrectly—preventing the very tasks that engineers find most frustrating.

Luckily, a small number of objections tend to be raised most of the time, and each of these objections has a straightforward response. In the end, it is usually not hard for a project manager to show most reasonable people that inspections are worth doing.

The most effective way a project manager can sell inspections to the organization is to show the savings in terms of time and money. Each inspection yields defects that would have been much more expensive to fix had they not been found; it should not be hard to give a rough idea of just how much time and money would have been wasted on those defects.

Another way the project manager can sell inspections is by pointing out the knowledge transfer benefits. By instituting inspections and code reviews, engineers other than the author of a work product are cross-trained on it, and can maintain it in the future if the author is busy with another project or has left the organization. Another way a project manager can help people accept inspections and understand their benefit is to run the first inspection meetings using work products created by people who are widely respected in the organization. Once others see the inspections run well and respectfully, they will be much more likely to accept the same practice applied to their own work.

When a project manager starts working toward implementing inspections, there are three objections that come up most often: people feel that inspections take too long, they do not like their work criticized, and they are protective of the final product. Luckily, it is not hard to anticipate these objections and give effective responses. (See Chapter 9 for more advice on making changes in an organization.)

"Inspections take too long."

Some team members seem to be opposed to anything that seems "bureaucratic." To them, inspections are just paper-shuffling meetings that waste their time. They should be writing code (or design specifications, test plans, requirements, etc.), and don't have time to waste just so some manager can check some box somewhere.

To convince someone with this mindset that inspections are necessary, the project manager must show him that every minute spent doing inspections can save many more down the road. Over the years, software engineering researchers have studied thousands of software projects in many different kinds of organizations. They have found again and again that a defect that takes a few minutes to fix in a vision and scope or a use case document will require hours, days, or weeks to fix in code or testing.

The project manager should explain that a typical inspection meeting will take less than two hours. If each person at the meeting finds a single defect, it more than makes up for the time that he spent reading and correcting the document. When looked at from this perspective, doing the inspection saves time.

"I don't like people criticizing my work."

Many people are uncomfortable putting their work up for review. They are unsure of their documents, and they are not used to having people point out their mistakes. It is very important to recognize this, because it often falls on the project manager to help the team members get comfortable with having their work put under a magnifying glass.

When this objection is raised, the project manager can point out that everyone makes mistakes, and that usually those mistakes are not the fault of the author. Frequently, when there is a problem in a document, it is because the author did not have enough information: bringing in the rest of the team can fill in those gaps.

The project manager can also point out that while it may be uncomfortable to have mistakes pointed out during the inspection, it's much more uncomfortable when those mistakes are left in the document. The author of a use case document may feel bad momentarily when defects are pointed out and corrected during an inspection meeting. But if he feels bad then, he will feel terrible if those defects are not caught until after the team spends months designing, programming, and testing the software, only to discover a "bug" that turns out to be a problem in his use case document. Instead of feeling bad when the inspection team points out problems, he should feel relieved that they were caught before they could cause project delays.

Defects should be discussed in terms of what is best for the work product or the project, not as criticisms of the author. It is very important that the moderator be extremely strict during the inspection meetings toward people who make rude personal comments. The moderator should enforce professionalism, and should ensure that every inspection meeting is conducted in a positive manner.

"I built it, and only I can say when it's done."

Some people are very protective of their work, and simply don't want other people to criticize it. In these cases, the author feels that she is the expert, and feels that there's nobody else in the organization who knows more about this subject than she does. Be very careful when confronting her—this is an emotionally charged situation, and it's very easy to turn this person off permanently. It is important for the project manager to be nonconfrontational. The project manager should work to influence this person, not to force her into a situation she doesn't want to participate in.

The best argument in this situation is to show her that the inspection is a tool that is there for her to use. It is like a spellcheck in a word processor: the document is always better after the spellcheck.

Nobody, no matter how good he is at his job, can deliver a perfect document. It is impossible to know exactly what's in the heads of the intended readers. There is no way to include the entire context in a document. There will always be technical or organizational concepts that would take pages and pages to explain, but that everyone is familiar with.

For example, a use case document for accounting software will not explain how double-entry accounting works; it will assume that every reader is familiar with the concept.

When an inspector finds a defect, he is helping the author identify areas that need to be explained. The author assumed that each reader fully understood a concept: it turned out that the reader needed some clarification after all. In this way, the document can be adjusted to the level of its specific readers.

The hesitant author will generally recognize that she has expertise that her readers do not have. Explain that while she can make an educated guess at what context and background needs to be included, she has no way of knowing if it is enough. The inspection process is an efficient technique to help her fix this.

> NOTE
>
> More information on inspections can be found in *Software Inspection* by Tom Gilb and Dorothy Graham (Addison Wesley, 1993) and *Peer Reviews in Software* by Karl Wiegers (Addison Wesley, 2002). Information on the effectiveness of software inspections can be found in the Software Technology Roadmap: *http://www.sei.cmu.edu/str/*.

Deskchecks

A *deskcheck* is a simple review in which the author of a work product distributes it to one or more reviewers. In a deskcheck, the author sends a copy of the work product to selected project team members. The team members read it, and then write up defects and comments to send back to the author. Work products that are commonly reviewed using a deskcheck include vision and scope documents (see Chapter 2) and discussion summaries (see Chapter 6).

There are times when a full inspection is neither necessary nor useful. Some work products do not benefit enough to warrant the attention of an entire inspection team because they do not need consensus or approval. In these cases, the author simply needs input from others to prevent defects, but does not require that they approve the document. In these cases, the deskcheck is a useful review practice.

Unlike an inspection, a deskcheck does not produce written logs that can be archived with the document for later reference. There is no follow-up meeting or approval process. It is simply a way for one team member to check another's work. Deskchecks are not formal reviews (where "formal" simply means that it generates a written work product that meets a certain standard and is archived with the rest of the project documentation); there is no standard for the results of the deskcheck. The reviewers simply review the work product and return the results. There is no moderator, and there is not necessarily any consensus generated.

But, despite the lack of formality, the deskcheck is a very important tool for a project team, and there are many times when the project manager will build deskchecks into the organization's software process. If a work product does not need approval by a team but is

still a critical part of the software process, the project manager may require a deskcheck in order to ensure that it does not have defects. For example, many QA teams employ automated test scripts, and it is usually necessary to ensure that the finished automated product actually covers the test plan that it was meant to automate. However, it would be unnecessary and very time-consuming to ask programmers, requirements analysts, project managers, and stakeholders to cross-reference each script with a test plan. A deskcheck can be used to verify that the script is correct, and to ensure that more than one QA engineer has taken responsibility for the quality of the script.

Sometimes a checklist is used to ensure that the work product meets the organization's standards. However, unlike an inspection, a deskcheck can be performed without a checklist. The deskcheck usually relies entirely on the reviewer's knowledge of the project and professional standards for the work product.

Figure 5-2 contains an example of comments from a deskcheck that was used by a tester to find defects in an automation script. In this case, the entire review was performed via email: the author mailed the script to the reviewer, and the reviewer read it and emailed the comments back to the author. These comments are much simpler than the inspection log in Figure 5-1. In an inspection, each log entry must either resolve a defect or indicate that it is an open issue that must be resolved. Deskcheck comments can simply point out issues or raise questions without having to supply solutions or promise a resolution. There was no follow-up or approval, and the reviewer had no more contact with this script.

Reviewer's name:	Sophie (senior QA engineer)
Author's name:	Dean (junior QA engineer)
Title:	Contract certification-automated test script #TP-491-A
Review date:	8/12/03
No. of review hours:	2

Location	**Comments**
Global	Script does not adequately copy databases in when the data changes.
Case 14	The test plan logs in as "Administrator;" this script logs in as "Admin."
Case 52, 53	What exactly is printed? It's not clear, you should be looking for specific data.
Case 61	The test plan tests all of the preferences, but the script only tests the first five.

FIGURE 5-2. Sample deskcheck comments

Deskchecks can be used as predecessors to inspections. In many cases, having an author of a work product pass his work to a peer for an informal review will significantly reduce the amount of effort involved in the inspection. Many defects can be caught by a single person reviewing a document. Approval and consensus is built later on during the inspection meeting; this is simply a way of saving effort. After a deskcheck, many authors will feel much more comfortable sending their document into an inspection—it will often help the author to be more objective and to take the inspection comments less personally.

Finally, a deskcheck can be useful to review a work product that is not meant to be inspected at all. For example, many requirements analysts will generate a discussion summary after a series of interviews and elicitation sessions (see Chapter 6). This is not a work product that is used in later stages of the software process; rather, it is an intermediate document used to generate the software requirements specification. A deskcheck is useful in this case to help interviewees and other requirements analysts identify any information gathered during the interviews that is inaccurate or unclear. No approval is needed, and the requirements analyst is free to ignore any of the comments. The deskcheck simply serves as a checkpoint to ensure that mistakes are caught and addressed as early as possible.

> NOTE
> **More information on deskchecks can be found in *Peer Reviews in Software* by Karl Wiegers (Addison Wesley, 2002).**

Walkthroughs

A *walkthrough* is an informal way of presenting a technical document in a meeting. Unlike other kinds of reviews, the author runs the walkthrough: calling the meeting, inviting the reviewers, soliciting comments, and ensuring that everyone present understands the work product. It typically does not follow a rigid procedure; rather, the author presents the work product to the audience in a manner that makes sense. Many walkthroughs present the document using a slide presentation, where each section of a work product is shown using a set of slides. Work products that are commonly reviewed using a walkthrough include design specifications (see Chapter 7) and use cases (see Chapter 6).

Walkthroughs are used when the author of a work product needs to take into account the perspective of someone who does not have the technical expertise to review the document. For example, a requirements analyst must make sure that the use cases she builds will provide the functionality that the users need, but the user representatives may not have seen use cases before and would be overwhelmed by them. If these users are simply included as part of an inspection team, it is likely that they will read the document and, failing to find many defects, sit silently through the inspection meeting without contributing much. This is not their fault—their training is in the business of the organization, not in reading and understanding software engineering documents. This is where a walkthrough can be a useful technique to ensure that everyone understands the document.

Before the walkthrough, the author should distribute any material that will be presented to each person who will be attending. For example, if the walkthrough is done as a slide presentation, copies of the slides should be emailed to the attendees. If only a portion of that material is going to be covered, that should be indicated as well.

During the walkthrough meeting, the author should solicit feedback from the audience. This is an opportunity to brainstorm new or alternative ideas, and to check that each person understands the document that is being presented. The author should go through parts of the document to make sure that it was presented in as clear a manner as possible.

These guidelines can help an author lead a successful walkthrough meeting:

- Verify that everyone is present who needs to review the work product. This could include users, stakeholders, engineering leads, managers, and other interested people.
- Verify that everyone present understands the purpose of the walkthrough meeting and how the material is going to be presented.
- Describe each section of the material to be covered by the walkthrough.
- Present the material in each section, ensuring that everyone present understands the material being presented.
- Lead a discussion to identify any missing sections or material.
- Document all issues that are raised by walkthrough attendees.

After the meeting, the author should follow up with individual attendees who may have had additional information or insights. The document should then be corrected to reflect any issues that were raised.

> NOTE
> **Additional information on walkthroughs can be found in *Peer Reviews in Software* by Karl Wiegers (Addison Wesley, 2002)**

Code Reviews

A *code review* is a special kind of inspection in which the team examines a sample of code and fixes any defects in it. In a code review, a defect is a block of code that does not properly implement its requirements, that does not function as the programmer intended, or that is not incorrect but could be improved (for example, it could be made more readable or its performance could be improved). In addition to helping teams find and fix bugs, code reviews are useful for both cross-training programmers on the code being reviewed and for helping junior developers learn new programming techniques.

Select the Code Sample

The first task in a code review is to select the sample of code to be inspected. It's impossible to review every line of code, so the programmers need to be selective about which portion of the code gets reviewed. Many teams have found that it takes about two hours to review 400 lines of code (in a high-level language such as Java), although this estimate differs dramatically from team to team and depends on the complexity of the code being reviewed. At that rate, there is no way a team could review all of the code for a software project. Nor would the team want to—in any program, there is a good deal of uninteresting code that looks very similar to the code already developed in previous applications, which has a lower risk of containing as many defects.

The purpose of any inspection is to find and repair defects. Since a relatively small portion of the code will be reviewed, it's important to review the code that is most likely to have defects. This will generally be the most complex, tricky, or involved code.

There are a few useful rules of thumb that are helpful:

- Is there is a portion of the software that only one person has the expertise to maintain? That may be a good candidate for review, for two reasons. First, because the rest of the team will learn how to maintain it; second, it's only ever been looked at by one person, so nobody else has yet had a chance to catch any defects in it.
- Does the software implement a highly abstract or tricky algorithm? The more difficult the algorithm, the more likely it is that a programmer introduced errors in its implementation.
- Is there an object, library, or API that is particularly difficult to work with? Working with a nonintuitive interface causes many programmers to make mistakes.
- Was the code written by someone who is inexperienced or has not written that kind of code before? Does it employ a new programming technique? Is it written in an unfamiliar language? A programmer who is doing something for the first time is most likely to introduce errors.
- Is there an area of the code that will be especially catastrophic if there are defects? A core tax accounting function is more important than the code that renders the splash screen. Select code that must not fail so that more people can look at it—and will be able to maintain it if it does have problems.

It is important to select a sample of code that an inspection team can review in about two hours. The project manager should try to keep the meeting to two hours or less, to avoid "meeting fatigue."

Hold the Review Session

The team selection and preparation in a code review are similar to any other kind of inspection. An inspection team of 3 to 10 people must be selected. Each of these people must be technically capable of reading and understanding the code being reviewed. Before the meeting, the moderator distributes the code sample to each inspector, who does individual preparation exactly as in the inspection.

The main difference between a code review and any other kind of inspection is in the review session. While the code review session is similar to the inspection meeting (see "Page-by-page review" above), there are a few important differences.

In addition to the moderator, there is a code reader who reads the code aloud during the meeting. The code reader can also be one of the inspectors; the only requirement is that the reader must have enough technical expertise to understand the code. The purpose of the reader is simply to keep the team's place during the inspection; the team should have already read the code and identified defects during their preparation. Since code is usually organized in logical units or blocks, it is more useful for a reader to go through those, rather than having the moderator go through the document page by page.

The reader starts at the beginning of the code sample and announces the first block or logical unit. She does not literally read the commands in the code; she simply gives a brief description (about one sentence) of the purpose of that block. If anyone (including the

reader) does not understand what the code does or disagrees with the interpretation, the author of the code explains what it is the code is supposed to accomplish. Sometimes the team can suggest a better, more self-explanatory way to accomplish the same thing; often it is simply a matter of explaining the purpose of the code to the person who raised the issue. If any inspectors found a defect in that block of code, the issue is raised and the team either comes up with a fix on the spot or tags it as an open issue for the programmer to fix later. The moderator then updates the inspection log, and the inspection continues until the reader completes the code sample being inspected.

Another important difference between code reviews and document inspections is that the code review is much more focused on detecting defects, and less on fixing them. This is because many defects in documents can be corrected with one or two sentences, or with a change in wording. Defects in the code can be much more involved, and there are often many ways that they could be fixed. The discussion of each defect is longer in a code review than it is during an inspection, and there are usually many open issues at the end of the code review.

In the code review, the moderator needs to be especially careful not to let the meeting turn into a problem-solving session. Programmers love to solve problems. It's easy for them to get caught up in a small detail and turn the meeting into an analysis of a minute problem that covers just a few lines of code. However, long discussions like this will prevent significant amounts of code from being reviewed. That's not to say that these discussions are not valuable—they just don't belong in the code review meeting. If a discussion looks like it will take more than three minutes or so, the moderator should stop the discussion and add it to the inspection log as an open issue. There should be few open issues, however, because most code defects should be straightforward to describe and document once they have been identified.

There are effective ways to modify the code during the review. Many inspectors have found that it is very helpful to refactor the code during the review. By applying refactorings on the spot, the team can make the code much more readable and identify additional defects. (See Chapter 7 for more information on refactoring.)

After the inspection meeting, the code author performs the rework and closes the open issues, and the moderator follows up with each of the inspectors and gains their approval. Instead of getting formal sign-off with physical signatures, it is usually sufficient to indicate the approval in the log comments when the changes are committed to the version control system (see Chapter 7 for more information on version control).

There are several additional benefits for the code review. One is that people learn how their teammates think about the code. A good way to encourage this is to switch off code readers in each review, so every team member gets a chance to be a reader. Reading code aloud and explaining it helps programmers think through problems. Every programmer should be able to explain his ideas well; discussing code during a code review is good practice for that.

Another benefit is that people who know that their code may be inspected tend to write more maintainable software. It's very common for programmers to not include comments or to write very terse, confusing code when they know that they are the only people who will ever read it. But if a programmer knows that someone else will be looking at it, he may put a lot of effort into making it readable. This can have enormous savings in maintenance efforts down the road.

Code Review Checklist

The following attributes should be verified during a code review:

Clarity

Is the code clear and easy to understand?

Did the programmer unnecessarily obfuscate any part of it?

Can the code be refactored to make it clearer?

Maintainability

Will other programmers be able to maintain this code?

Is it well commented and documented properly?

Accuracy

Does the code accomplish what it is meant to do?

If an algorithm is being implemented, is it implemented correctly?

Reliability and Robustness

Is the code fault-tolerant? Is it error-tolerant?

Will it handle abnormal conditions or malformed input?

Does it fail gracefully if it encounters an unexpected condition?

Security

Is the code vulnerable to unauthorized access, malicious use, or modification?

Scalability

Could the code be a bottleneck that prevents the system from growing to accommodate increased load, data, users, or input?

Reusability

Could this code be reused in other applications?

Can it be made more general?

Efficiency

Does the code make efficient use of memory, CPU cycles, bandwidth, or other system resources?

Can it be optimized?

Pair Programming

Pair programming is a technique in which two programmers work simultaneously at a single computer and continuously review each others' work. Although many programmers were introduced to pair programming as a part of Extreme Programming (see Chapter 12), it is a practice that can be valuable in any development environment. Pair programming improves the organization by ensuring that at least two programmers are able to maintain any piece of the software. Pair programming also helps programmers' professional development, because they learn from each other.

Pair programming is like having a continuous code review, without the extra time or effort of holding individual code reviews. It encourages a redundancy among the team members, and everyone is cross-trained on various parts of the code. Junior people can more quickly learn directly from senior people when they are paired together. While it may seem that assigning two people to a single programming task could be inefficient, in fact, productivity often increases. It takes somewhat less time to perform each task, as there are often gaps where one person is "tapped out" and the other can take over. More importantly, the resulting code is of very high quality, so there are far fewer mistakes to go back and fix.

One useful benefit of pair programming is that people tend to write better code when they know that someone else will be reading it. They cut fewer corners, spend more time making the code readable, are more likely to include comments where necessary, and refactor more often.

In pair programming, two programmers sit at one computer to write code. Sometimes they share a single keyboard and mouse, although it is possible to get special hardware or cables that allow each programmer to have his own. Generally, one programmer will take control and write code, while the other watches and advises. But different pairs of people may discover their own dynamic: for example, some pairs will take turns at the keyboard, while others will designate one person as the typist (if one person types significantly faster than the other).

It's straightforward to implement pair programming in any development team—just choose two programmers who are willing to give it a shot, and have them work together at the same computer. However, it's important to remember that, like any programming technique, pair programming is a skill that improves with practice. Some benefits can be realized almost immediately, but there is no substitute for years of experience. But, like any other programming skill, the only way to get experience is to practice.

While efficient pair programming is a skill that requires practice and patience, there are some useful tips that make its initial adoption easier. People will be less resistant to a change if their first experience with a new technique is positive. One way to help guarantee this is to pilot pair programming on a low-risk portion of code. The project manager should choose one where the scope and requirements are well understood going into the project, and where success is easily measured. Both members of the pair assigned to work

on it should be people who have done similar projects in the past. These circumstances can provide an opportunity for an easy win, which in turn will increase the programmers' confidence in the technique.

Some teams have found that pair programming works best for them if the pairs are constantly rotated; this helps diffuse the shared knowledge throughout the organization. Any two programmers can potentially make a well-functioning pair, no matter what their relative experience. Some people have found that it helps to choose pairs that include both a senior person and a junior person. This will make it easier for the communication to fit into an existing pattern (mentor and tutor, roles that both people are already used to—although this is not necessarily how all senior-junior pairs will interact). Often, a junior team member will ask a seemingly "naïve" question about the code that turns out to identify a serious problem. This is especially common with problems that the senior member has been living with for so long that she no longer notices them. Sometimes, the extent of a code problem only becomes clear when it is explained to somebody else.

Pair programming is not for everyone. It is difficult to implement pair programming in an organization where the programmers do not share the same 9-to-5 (or 10-to-6) work schedule. Some people do not work well in pairs, and some pairs do not work well together. The project manager should not try to force pair programming on the team; it helps to introduce the change slowly, and where it will meet the least resistance. Some programmers will argue that assigning two people to one task is a waste of time, claiming that two people can get twice as much work done if they work separately. While this may seem true at first glance, the pair will introduce far fewer defects; it may require more man-hours to do the programming, but it will reduce the amount of time spent on bug-fixing and maintenance. However, this may not convince some stubborn programmers. In this case, an effective way to introduce this technique is to begin with the people who are more excited about the idea. Their success can help to convince the stragglers of the value of pair programming.

> NOTE
> **More information on pair programming can be found in *Extreme Programming Explained* by Kent Beck (Addison Wesley, 1999).**

Use Inspections to Manage Commitments

A successful project needs more than just a blanket agreement between team members. It's very easy for someone to "agree" to a document, only to turn around later and decide that he didn't fully understand what he was agreeing to. Instead, the project team needs to reach a true consensus, where each person fully supports the document. The goal of an inspection is to build consensus on the document by gaining a real commitment from everyone who has read it. When a reviewer approves a document, he takes responsibility for its contents, and if the document has defects, he shares some of the blame for missing the mistake.

The best way to reach consensus among the inspection team is for each person to feel like he or she made a real contribution to the document. The inspection meeting accomplishes that by allowing each person to find problems in the document and help the rest of the team find a solution to each problem. This is why it's important for the team to go beyond just pointing out the defects in the document and actually come up with replacement wordings that fix the defects.

By the end of the meeting, nobody remembers that one person suggested this sentence, and another suggested that one; everyone feels a sense of ownership because it was a real group effort. That ownership means that each team member leaves the meeting with a real commitment to the document. This can eliminate many of the conflicts that can cause problems later on in the project.

Inspections are also important for gaining real, meaningful approval for a document. When a document is not correct, that puts the team members in a very difficult situation. Consider a stakeholder who is reading a vision and scope document, and who has a problem with some of the contents. She can't just refuse to approve the document; that would mean that she was personally holding up the project. But if she lets the document move forward as is, that could cause serious problems later in the project. So the stakeholder feels backed into a corner. She can't approve the document as it stands, but she doesn't have the authority to make changes to it. Typically, people who are put in this situation simply avoid reading the document, giving them a sort of "plausible deniability" that lets them avoid blame when the project has problems later.

Inspections are a way out of this situation. The inspection team is given the responsibility of approving the document. To accomplish that, each member is given the authority to withhold approval until any text in the document that prevents them from approving it has either been changed to meet their needs or has been explained to the approver's satisfaction. This allows the project to move forward.

Other kinds of reviews are also useful in managing commitments. Deskchecks are especially important for gathering consensus among people in the organization who do not need to approve specific documents, but whose input is still very important.

Project teams are made up of people who all share a common goal: getting the software project out the door. Stakeholders, users, engineers, and project managers all have this goal in common. This means that each person should be willing to take on responsibility to make sure that every document produced over the course of the software project is correct.

Diagnosing Review Problems

Many organizations rely on their testers (or, in worse cases, their users) to find the bulk of the defects in the software they produce. When the defects are caused by simple coding errors or typos, they are easy to correct. Unfortunately, very few defects are caused by simple coding errors or typos. Most defects are introduced before a single line of code is written. Sometimes a programmer misunderstands the design; at other times, the entire

team fails to take a stakeholder's needs into account and fails to build a needed feature into the software. Waiting until after the software is built to discover these problems results in an enormous amount of work to fix defects that should have been caught before a single line of code was written.

Problems Are Found Too Late

There are many problems that can be avoided by having the team adopt vision and scope documents, project plans, software requirements specifications, and other project documents. But what happens when the team doesn't catch an error in one of these documents until the software is built?

One of the most common causes of project failure is that requirements contain a defect that is not caught until much later in the project. Trying to fix that defect after the software is built can be so costly that it can destroy the project entirely. For example, suppose that a team member writes a use case document to describe a critical feature. The document is emailed around to the team, and everyone reads it. However, some of the readers are very busy, so they only skim it and see that it looks about right. Others see problems, but don't want to embarrass the author by bringing them up. A few think that they found very obscure problems and don't want to embarrass the other readers who they think would not have come up with the problem.

Some teams try to find defects by mailing documents around to the team, with no real expectations of what the team is supposed to do with each document. After the author mails the document out, usually nobody responds to the email for a few days. Responses trickle in over the course of the following week. Some of them point out serious flaws, but most of them do little more than point out typos and minor wording changes. In the meantime, the designers start their work to avoid sitting around doing nothing. Eventually the user interface, architecture, and software are built. The product is passed down to the QA team, who start testing the build. They haven't been a part of the development of the software at all, and have only been talking to the users and stakeholders, putting together a test plan to ensure that the software does everything that they expect it to. Within a few days, the QA team discovers a problem: there is a feature that does not work the way the users need it to work.

When this happens, the entire project team is brought together in a meeting, and a split quickly forms. The QA lead and stakeholder insist that the software is broken. The requirements analyst, design lead, and programming lead insist that there is nothing wrong. They finally start going through the documents and find that the software does, indeed, meet all of the requirements laid out in the relevant use case. But the use case is wrong—it does not describe what the users need the software to do. It's clear to everyone present that fixing the problem will be a major endeavor, and that the project will be delayed for a very long time. The team built software that was solid, well-built, and functional—but they built the wrong software.

If the project team had inspected the use cases, this could have been avoided. Everyone with a stake in the project—including QA team members—would be invited to the inspection meetings, and each inspector would have a better idea of what to look for during the inspection process. It costs the same to build the right software as it does to build the wrong software. A few hours of searching for and fixing any problems with the use case document would have saved the team weeks or months of rework.

Big, Useless Meetings

Having a project fail due to a problem in a document that isn't caught until late in the project is traumatic for a team. It's especially bad for the person who wrote the document. Once a team experiences this problem, everyone feels especially motivated to do something about it. In many cases, the solution that seems most obvious to the team and the project manager is to distribute the responsibility for creating the document. The last project was a mess because the team missed something; there's no way that they will let this happen again.

The project manager calls a meeting to get everyone together at the very beginning of the project. He takes no chances, inviting everyone who might possibly have some small input into the project and impressing upon everyone just how important the meeting is. An entire afternoon is blocked out for a standing-room-only session that's supposed to let everybody have a voice in the design of the document. Unfortunately, it ends up having the opposite effect.

Everyone has something to say, and nobody wants to let any stone go unturned. The big meeting seemed like a good idea on paper, but it quickly gets bogged down in details that only one or two people care about. The meeting goes nowhere. Nobody knows what they are supposed to do, and there's no real leadership or direction. Long after meeting fatigue sets in, no progress has been made. Stakeholders are arguing about the minute details of the day-to-day business of the organization, while designers are caught up in potential design problems, programmers can't decide on which tools to use, and, through all of this, nothing is written down. The team adjourns the meeting, having made no progress. After a few tedious marathon meetings, the project team gets sick of waiting and decides to do things the way that they've always been done.

Had the team held an inspection meeting, a deskcheck, or a walkthrough, everyone would have understood his or her role. The reviewers would have been more carefully selected. A single lead author would have had the responsibility of generating the document. Each reviewer would have had a well-defined role, reading the document and bringing up specific defects. Each defect would have been discussed by people with some knowledge to address it, and the responsibility for finding the errors would rest with the people capable of fixing them.

The Indispensable "Hero"

Sometimes a programming team has one "hero" who seems to stand out above everyone else. If a technical problem comes up that nobody can solve, and it looks like the deadline

will be blown, the hero will often take the problem home on Friday night, work all weekend, and come in on Monday morning with a solution.

It seems like the hero is good for the team. But there are some serious downsides to his heroics. He's a constant scheduling problem for the project manager, because it seems that no project can be completed without him. He's constantly over-allocated, and there are entire programming teams who cannot move forward because they are waiting for him to finish a project. Meanwhile, he is constantly working 70-hour weeks, and the entire team is afraid that he will burn out or leave the organization.

In some cases, the hero is inadvertently keeping the rest of the team from advancing, either professionally or in the organization. It seems that the hero wrote the core of every code library. Only he knows the details of critical architecture pieces. The hero is tired of people talking about him getting hit by a bus, which seems to come up at least once in every architecture or code planning meeting.

The most difficult problem to deal with in this situation is maintenance. Because of his peculiar over-allocation problem, there is an increasing amount of code that only the hero is able to maintain. This is usually because he was called in to write the most difficult part of the code. Sometimes it is algorithmically difficult, and he's the only one with enough experience to do it. At other times, the coding task relies on a library that he wrote and that only he knows how to use. In either of these cases, if that code needs to be updated, the hero is the only person in the organization who is familiar with it.

Code reviews and pair programming can help alleviate the dependence on the hero. When he writes a piece of especially tricky code, he can hold a code review. If a group of programmers inspects that code, they will be able to maintain his code. In future projects, they'll be able to draw on what they learned in the review session. This will help the entire team's professional development. What's more, the hero is often not much more advanced than everyone around him; he may just know a few tricks that the rest of the team can begin to pick up. Pair programming can be especially helpful if the hero is teamed up with another senior programmer. Sometimes the "hero" status is merely a matter of perception—everyone just "knows" that he's the best programmer around. Pair programming can help everyone on the team realize that they have other people who are just as valuable. For the true hero, sharing his skills with others will help him earn real respect from the team. Team members will be able to continue to learn from his experience, and he will be able to share and teach the team. The team, in turn, will come to see him as a role model and a leader, instead of just a hero who swoops in to fix their problems.