Chapter Three. Peer Review Formality Spectrum

I have been using the terms “review” and “peer review” as synonyms meaning any manual examination performed to look for errors in a software deliverable. However, there are several distinct review approaches. The software literature contains conflicting definitions and inconsistent usage for the names of these activities. For example, “walkthrough” has been used to describe many types of reviews, ranging from casual group discussions to formal inspections. One organization changed its walkthrough process to an inspection process simply by globally replacing the word “walkthrough” with “inspection.” It really was an inspection process, although the original authors had mistakenly used the walkthrough terminology.

This chapter describes several kinds of peer reviews that span a range of formality and rigor. It also suggests ways to select an appropriate review technique for a given situation. You might be used to different terminology, but I’m presenting these descriptions here to provide a common vocabulary for this book.

The Formality Spectrum

Peer reviews can be classified based on their degree of formality or their relative levels of discipline and flexibility (Iisakka, Tervonen, and Harjumaa 1999). Figure 3–1 places several common review methods along a formality scale. These are not the only types of technical peer reviews, and multiple variations of many of them exist. The most formal reviews, such as inspections, have several characteristics:

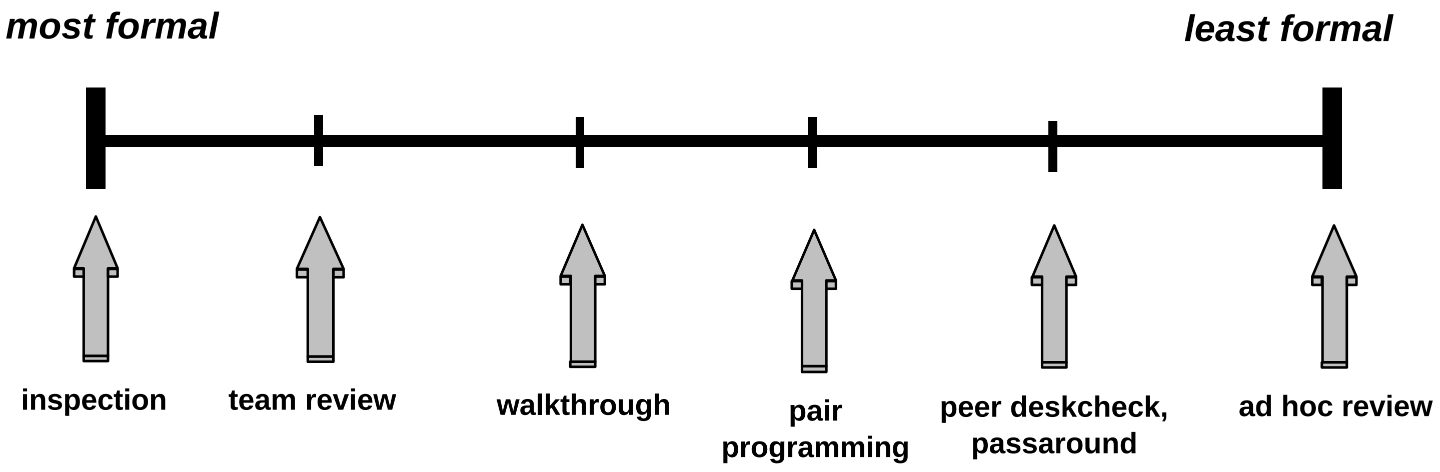

Figure 3-1. Peer review formality spectrum.

- Defined objectives
- Participation by a trained team
- Leadership by a trained moderator
- Specific participant roles and responsibilities
- A documented review procedure
- Reporting of results to management
- Explicit entry and exit criteria
- Tracking of defects to closure
- Recording of process and quality data

Informal reviews may well meet your needs in certain situations. They are quick and cheap, do not require advance planning, demand no organizational infrastructure, and can help the author proceed on an improved course. Learn about the strengths and limitations of the various approaches so you can select a review process for each situation that fits your culture, time constraints, and business and technical objectives. I recommend that you begin performing inspections at the outset to gain experience so you can judge when a less formal review approach is appropriate and when it is not. If you start by holding just informal reviews with the intention of moving to inspections someday, you'll miss out on the full advantages of inspection.

All peer reviews involve some combination of planning, individual examination of the work product, review meetings, correction of errors, and verification of the corrections. Table 3–1 shows which of these activities are typically included in each of the review types shown in Figure 3–1.

Table 3–1. Activities Typically Included in Different Types of Peer Reviews

	Activity				
Review Type	***Planning***	***Preparation***	***Meeting***	***Correction***	***Verification***
Inspection	Yes	Yes	Yes	Yes	Yes
Team Review	Yes	Yes	Yes	Yes	No
Walkthrough	Yes	No	Yes	Yes	No
Pair Programming	Yes	No	Continuous	Yes	Yes
Peer Deskcheck, Passaround	No	Yes	Possibly	Yes	No
Ad Hoc Review	No	No	Yes	Yes	No

Inspection

An *inspection* is the most systematic and rigorous type of peer review (Ebenau and Strauss 1994; Fagan 1976; Gilb and Graham 1993; Radice 2001). The term “formal inspection” is redundant but is sometimes used for emphasis. Inspection has been identified as a software industry best practice, while less formal review approaches have not earned this status (Boehm and Basili 2001; Brown 1996; McConnell 1996). Inspection follows a multistage process with specific roles assigned to individual participants. Although several variations on the inspection theme have been developed, their similarities outweigh their differences. Chapter 4 describes a common inspection process that includes seven stages: planning, overview, preparation, meeting, rework, follow-up, and causal analysis.

For maximum effectiveness, inspectors should be trained in the inspection process so they can perform the various participant roles. Inspections rely on checklists of defects commonly found in different types of software work products and other analytical techniques to search for bugs. Some reasons to hold inspections include the following (IEEE 1999b):

- To verify that a product satisfies its functional specifications, specified quality attributes, and customer needs and to identify any deviations from these
- To verify that a product conforms to pertinent standards, regulations, rules, plans, and procedures and to identify any deviations from these
- To provide metrics on defects and inspection effort that can lead to improvements in both the inspection process and the organization’s software engineering process (see Chapter 9)

An important aspect of an inspection is that participants other than the work product’s author lead the meeting (inspection role of *moderator*), present the material to the inspection team (*reader*), and document issues as they are brought up (*recorder*) (IEEE 1999b). Compared to other reviews, inspection provides the most thorough coverage of the work products. The participants prepare for the inspection meeting by examining the material on their own. During the meeting, the reader presents one small portion of the material at a time to the other inspectors, who then raise issues, ask questions, and point out possible defects. Because several inspectors carefully scrutinize the product, inspection provides a good test of understandability and maintainability. The use of a reader helps the team reach the same interpretation of each portion of the product, because the inspectors can compare their understanding to that expressed by the reader. At the end of the inspection meeting, the team agrees on an appraisal of the work product and judges whether changes the author will make during rework must be verified through a second inspection, by a single person, or not at all.

Inspections are more effective at finding defects than are informal reviews. One telecommunications company detected an average of 16 to 20 defects per thousand lines of code by inspection, compared to only three defects per thousand lines of code when using informal reviews. Inspections are especially important for high-risk products that must be as free of defects as possible. The close scrutiny of an inspection helps reveal programming problems such as off-by-one errors, incorrect arguments in function calls, missing cases, and situations in which one routine will cause problems in another.

Different inspectors spot different kinds of problems, but this contribution does not increase linearly with additional participants. Many defects are found redundantly. When one inspector notes a defect, it's common to hear another participant say, "I saw that, too." The interactions between inspectors during the meeting can reveal new problems as one person's observation stimulates another's thinking. Michael Fagan, who developed the best-known inspection method, termed this synergy the Phantom Inspector (Fagan 1986). Although some studies have questioned whether synergy adds significant value, a peer review that does not include a meeting loses this potential collaborative benefit.

Team Review

Team reviews are a type of "inspection-lite," being planned and structured but less formal and less rigorous than inspections. A team review allows a group of qualified people to judge whether a product is suitable for use and to identify ways in which the product does not satisfy its specifications. The team review goes by many names, often simply being called a "review." The structured walkthrough approach devised by Edward Yourdon is similar to what I term a team review (Yourdon 1989). Team reviews cost more than having a single colleague perform a peer deskcheck, but the different participants will find different problems. A team review provides a good learning opportunity for the participants.

As I have practiced them, team reviews follow several of the steps found in an inspection. The participants receive the review materials several days prior to the review meeting and are expected to study the materials on their own. The team collects data on the review effort and the defects found. The overview and follow-up inspection stages are simplified or omitted, however, and some participant roles may be combined. As with any meeting, the team can get sidetracked on tangential discussions, so a moderator is needed to keep the meeting on course. A recorder or scribe captures issues as they are raised during the discussion, using standard forms the organization has adopted.

The guidelines and procedures described in this book for inspection also apply to team reviews except in the way the review meeting is conducted. The author might lead a team review, whereas in an inspection the author is not permitted to serve as the moderator. In contrast to most inspections, the reader role is omitted. Instead of having one participant describe a small chunk of the product at a time, the moderator asks the participants if they have any issues on a specific section or page.

Although little published data is available, one industry study found that this type of team review discovered only two-thirds as many defects per hour as inspections revealed (Van Veenendaal 1999). IBM Federal Systems Division measured coding productivity for projects that practiced structured walkthroughs (team reviews) at about half the productivity of similar projects that used inspections (Gilb and Graham 1993). Team reviews are suitable for a group or work product that doesn't require the full rigor of the inspection process. Being less formal than inspection, team reviews might also devote some meeting time to discussing solution ideas and having participants reach consensus on technical approaches.

Walkthrough

A *walkthrough* is an informal review in which the author of a work product describes the product to a group of peers and solicits comments (Hollocker 1990). Walkthroughs differ significantly from inspections because the author takes the dominant role; other specific review roles are usually not defined. Whereas an inspection is intended to meet the team's quality objectives, a walkthrough principally serves the needs of the author. Table 3–2 points out some differences between the ways that inspections, team reviews, and walkthroughs (as I'm defining them here) usually are performed.

Table 3–2. Comparison of Some Inspection, Team Review, and Walkthrough

Characteristics

Characteristic	***Inspection***	***Team Review***	***Walkthrough***
Leader	Moderator	Moderator or Author	Author
Material presenter	Reader	Moderator	Author
Granularity of material presented	Small chunks	Pages or sections	Author's discretion
Recorder used	Yes	Yes	Maybe
Documented procedure followed	Yes	Maybe	Maybe
Specific participant roles	Yes	Yes	No
Defect checklists used	Yes	Yes	No
Data collected and analyzed	Yes	Maybe	No
Product appraisal determined	Yes	Yes	No

Walkthroughs are informal because they typically do not follow a defined procedure, do not specify exit criteria, require no management reporting, and generate no metrics (Fagan 1986). They can be an efficient way to examine work products modified during maintenance, because the author can draw the reviewers' attention to those portions of the deliverables that were changed. However, this runs the risk of overlooking other sections that should have been changed but were not. Because records are rarely kept, there is little data about how effective walkthroughs are at detecting bugs. One report from Ford Motor Company indicated that inspections found 50 percent more defects per thousand lines of code than did walkthroughs (Bankes and Sauer 1995).

When walkthroughs do not follow a defined procedure, people perform them in diverse ways that range from casual to disciplined. In a typical walkthrough, the author presents a code module or design component to the participants, describing what it does in the product, how it is structured and how it performs its tasks, the logic flow, and its inputs and outputs. Finding problems is one goal. Another is reaching a shared understanding and agreement as to the module's purpose, structure, and implementation. Design walkthroughs provide a way to assess whether the proposed design is sufficiently robust and appropriate to solve the problem. Arguing the correctness and soundness of a proposed design leads to improvement as well as to defect detection.

Authors can use walkthroughs to test their ideas, brainstorm alternative solutions, and stimulate the creative aspects of the development process. Walkthroughs are a good way to evaluate test documentation. The author can lead the team through the test cases to reach a shared understanding of how the system should behave under specified conditions. Usability walkthroughs are a tool for evaluating a user interface design from a human factors perspective (Bias 1991).

A walkthrough is appropriate when a prime review objective is to educate others about the product. You can include more people in a walkthrough than can participate effectively in an inspection. However, walkthroughs don't let the participants judge how understandable the product is on its own. After the walkthrough, you aren't sure if you understood the material because it was well structured or because the author presented it clearly (Freedman and Weinberg 1990).

If you don't fully understand some information presented in a walkthrough, you might gloss over it and assume the author is right. It's easy to be swayed into silence by an author's rationalization of his approach, even if you aren't convinced. Sometimes, though, a reviewer's confusion is a clue that a defect lurks nearby or that something needs to be expressed more clearly. My colleague Kevin once worked with a "technical expert" who created designs that no one else could understand during walkthroughs. At first, Kevin's group assumed this developer was wiser than they were. Eventually, though, they realized that he just created convoluted designs. They tried using formal inspections to rein him in. Unfortunately, their manager was fooled by the "expert's" apparent brilliance and let him continue to design and code in a vacuum. Reviews won't help people who reject all suggestions from their colleagues.

In the past, the programmer often stepped through a module's execution during a walkthrough, using some sample data, with his peers checking for correct logic and behavior. Interactive debuggers are more commonly used for this purpose today. Another walkthrough strategy involves following a script that describes a specific task or scenario to illustrate how the system would function during a sample user session. This technique helps a group of customers, requirements analysts, and developers reach a common understanding of system behavior. It can also reveal errors in early deliverables such as requirements specifications. Walkthroughs have a place in your peer review tool kit, but the domination by the author and the unstructured approach render them a less valuable defect filter than inspections or team reviews.

Pair Programming

Pair programming is a component of a popular "agile methodology" approach to software development called Extreme Programming (Beck 2000). In pair programming, two developers work on the same program simultaneously at a single workstation. This approach facilitates communication and permits continuous, incremental, and informal review of each person's ideas. Every line of code is written by two brains driving a single set of fingers, which leads to superior work products by literally applying the adage "two heads are better than one." The member of the pair who is currently doing the typing is the driver, while the other is the partner (Jeffries, Anderson, and Hendrickson 2001). The driver and partner exchange roles from time to time.

Culturally, pair programming promotes collaboration, an attitude of collective ownership of the team's code base, and a shared commitment to the quality of each component (Williams and Kessler 2000). Two team members become intimately familiar with every piece of code, which reduces the knowledge lost through staff turnover. The pair can quickly make course corrections because of the real-time review by the partner. Rapid iteration leads to robust designs and programs. The pair programming technique can be applied to create other software deliverables besides code.

I classify pair programming as a type of informal review because it is unstructured and involves no process, preparation, or documentation. It lacks the outside perspective of someone who is not personally attached to the code that a formal review brings. Nor does it include the author of the parent work product as a separate perspective. Pair programming is not specifically a review technique. Instead, it is a software development strategy that relies on the synergy of two focused minds to create products superior in design, execution, and quality. Some evidence indicates that pair programming leads to higher product quality and accelerates completion of programs compared to the time needed by individuals (Williams et al. 2000). However, pair programming constitutes a major culture change in the way a development team operates, so it's not a simple replacement for traditional peer reviews in most situations.

Peer Deskcheck

In the early days of programming, we studied our source listings carefully between infrequent compilations to find errors in the hope of ensuring a clean execution. This is a *deskcheck*. Contemporary code editors and fast compilations greatly enhance developer productivity, but they have also made us less disciplined about reviewing our own work. Don't underestimate the value of careful deskchecks. Meticulously examining a printed listing reveals far more errors than you'll see by scanning the code on your monitor. Deskchecking is an integral aspect of the highly regarded Personal Software Process (Humphrey 1995). A deskcheck is a kind of self-review, not a peer review.

In a *peer deskcheck* (also known as a buddy check and pair reviewing), only one person besides the author examines the work product. The author might not know how the reviewer approached the task or how comprehensively the review was done. A peer deskcheck depends entirely on the single reviewer's knowledge, skill, and self-discipline, so expect wide variability in results. Peer deskchecks can be fairly formal if the reviewer employs defect checklists, specific analysis methods, and standard forms to keep records for the team's review metrics collection. Upon completion, the reviewer can deliver a defect list to the author, they can sit down together to prepare the defect list, or the reviewer can simply hand the author the marked-up work product.

The peer deskcheck is the cheapest review approach. It involves only one reviewer's time, which might include a follow-up discussion with the author to explain the reviewer's findings. This method is suitable if you have colleagues who are skilled at finding defects on their own, if you have severe time and resource restrictions, or for low-risk work products. A peer deskcheck can be more comfortable for the author than a group review; however, the errors found will only be those the one reviewer is best at spotting. Also, the author is not present to answer questions and hear discussions that can help him find additional defects no one else sees. You can address this shortcoming by having the author and the single reviewer sit down together in what has been termed a two-person inspection (Bisant and Lyle 1989). The two-person inspection lacks a moderator, and the one inspector also functions as the reader.

Peer deskchecks provide a good way to begin developing a review culture. Find a colleague you respect professionally and trust personally and offer to exchange work products for peer deskchecks. This is also a good mentoring method, providing it's done with sensitivity. Senior developers at Microsoft review code written by new people or by developers working in a new area (Cusumano and Selby 1995). In addition to providing a quality filter, such reviews provide a coaching opportunity to pass along tips for better ways to do things the next time. Be careful, though: there is a thin line between asking an old hand to help out a junior developer and implying that incompetent programmers need someone to second-guess their work.

Passaround

A *passaround* or distribution is a multiple, concurrent peer deskcheck. Instead of asking just one colleague for input, you deliver a copy of the product to several people and collate their feedback. This book was reviewed by using a passaround approach, with an average of about 16 people providing comments on each chapter. Some comments were superficial, while others led to significant restructuring and major improvement. As is common with a passaround, I never heard from several people who volunteered to participate, and others returned their comments too late to be useful. Late contributors to the passaround often are wasting their time reviewing an obsolete version of the work product. I also received some conflicting suggestions, which I had to reconcile.

The passaround helps mitigate two major risks of a peer deskcheck: the reviewer failing to provide timely feedback and the reviewer doing a poor job. At least some of your multiple reviewers will probably respond on time, and several will probably provide valuable input. You can engage more reviewers through a passaround than you can conveniently assemble in a meeting. However, a passaround still lacks the mental stimulation that a group discussion can provide. Once I used a passaround to have other team members review my development plan for a new project. Unfortunately, none of us noticed that some important tasks were missing from my work breakdown structure. We thought of these missing tasks later during a team meeting, which suggests that we would have found them if we had used a team review or inspection instead of the passaround.

As an alternative to distributing physical copies of the document to be reviewed, one of my groups placed an electronic copy of the document in a shared folder on our server. Reviewers were invited to contribute their comments in the form of document annotations, such as Microsoft Word comments or PDF notes, for a set period of time. Then the author reconciled conflicting inputs from the reviewers, making obvious corrections and ignoring unhelpful suggestions. Only a few issues remained that required the author to sit down with a particular reviewer for clarification or brainstorming.

This passaround method allows each reviewer to see the comments that others have already written, which minimizes redundancy and reveals differences of interpretation. Watch out for debates that might take place between reviewers in the form of document comments; those are better handled through direct communication. These document reviews are a good choice when you have reviewers who cannot hold a face-to-face meeting because of geographical separation or scheduling restrictions (see Chapter 12 for other ways to deal with these challenges). Several researchers have developed collaborative tools that extend this simple approach to asynchronous, “virtual” reviewing (Iisakka, Tervonen, and Harjumaa 1999; P. Johnson et al. 1993; P. Johnson 1994, 1996a; Mashayekhi, Feulner, and Riedl 1994). Appendix B contains pointers to some of these tools.

Ad Hoc Review

Chapter 1 opened with a story about one programmer asking another to spend a few minutes helping to track down an elusive bug. Such spur-of-the-moment reviews are a natural part of software team collaboration. They provide a quick way to get another perspective that often finds errors we just cannot see ourselves. Ad hoc reviews are the most informal type of review, having little impact beyond solving the immediate problem.

Choosing a Review Approach

One way to select the most appropriate review method for a given situation is to consider risk: the likelihood that a work product contains defects and the potential for damage if it does. Whether a defect is of major or minor severity depends on its context and the impact it could have if left uncorrected. A small logic error might be a cosmetic irritant if some text is the wrong color but literally fatal in a module that controls a life-support system. Reviews should focus on finding the most severe defects.

Studies have shown that large systems usually contain a small number of error-prone modules. In one system that contained 425 modules, 58 percent of all customer-reported defects resided in only 31 modules (C. Jones 1997). Nearly all of the defects will cluster in just half of the modules, and about 20 percent of the modules will contain about 80 percent of all defects (Boehm and Basili 2001). Factors that contribute to creating error-prone modules include excessive developer schedule pressure, inadequate developer training or experience, and creeping requirements that lead to many changes (C. Jones 1997). Other factors that increase risk include

- Use of new technologies, techniques, or tools
- Complex logic or algorithms that must be correct and optimized
- Mission- or safety-critical portions of the product with severe failure modes or many exception conditions, particularly if they are difficult to trigger during testing
- Key architectural components that provide a base for subsequent product evolution
- Components that are intended to be reused
- Components that will serve as models or templates for other components
- Components with multiple interfaces that affect various parts of the product

Combining informal reviews with inspections provides a powerful quality strategy. During requirements development on one project, we used a series of passarounds to have customer participants informally review the growing requirements specification. The passarounds revealed many errors quickly and cheaply. After we assembled the complete specification from portions written by several analysts, we held a formal inspection, again including key customers. We found an average of 4.5 additional defects per page. Had we not held the informal reviews, we would have had to deal with many more defects during the final inspection, and we certainly would have overlooked other problems in the defect-riddled specification. My quality philosophy is to review such key project documents early and often, both formally and informally, because the cost of finding requirements errors is so much lower in the early stages of the project.

One software quality objective is to reduce the risk associated with a given deliverable to an acceptable level. The project team should select the cheapest review method that will accomplish this goal. Use inspections for high-risk work products, and rely on cheaper techniques for components having lower risk. You can also choose a review technique based on the stated objectives for the review. Table 3–3 indicates which review approaches can help you achieve specific objectives. The best way to judge which peer review method to use in a given situation is to keep records of review effectiveness and efficiency in your own organization. You might find that inspections work well for code, whereas team reviews or walkthroughs are better for design documents. Hard data provides the most convincing argument, but even subjective records of the types of reviews held, the products examined, and the effectiveness of the reviews will be useful.

Table 3–3. Suggested Review Methods for Meeting Various Objectives

Review Objectives	Inspection	Team Review	Walkthrough	Pair Programming	Peer Deskcheck	Passaround
Find product defects	X	X	X	X	X	X
Check conformance to specifications	X	X			X	X
Check conformance to standards	X				X	X
Verify product completeness and correctness	X		X			
Assess understandability and maintainability	X	X		X		X
Demonstrate quality of critical or high-risk components	X					
Collect data for process improvement	X	X				
Measure document quality	X					
Educate other team members about the product		X	X	X		X
Reach consensus on an approach		X	X	X		
Ensure that changes or bug fixes were made correctly		X	X		X	
Explore alternative approaches			X	X		
Simulate execution of a program			X			
Minimize review cost					X	