Software Reviews and Pair Programming

Software reviews are a quality assurance technique that helps us remove defects from our software programs and supporting documentation. In this chapter, we will explain the following:

- the benefits of software reviews
- the review technique of personal reviews, walkthroughs, and formal inspection.
- the economics of software reviews

...three experienced engineers worked for three months to find a subtle system defect that was causing persistent customer problems. At the time they found this defect, the same code was being inspected by a different team of five engineers. As an experiment, this team was not told about the defect. Within two hours, this team found not only this defect, but also 71 others! Once found, the original defect was trivial to fix. (Humphrey, 1995)

By this point in your life, we’re sure you have written a paper and had someone else read it before you turned it in. Quite often, authors can be pleased with their own work and then quite shocked when others find mistakes or have excellent suggestions on how to make it even better. We’re not the best judges of our own work, and to make matters worse, we’re quite blind to our own mistakes. In the classic book, *The Psychology of Computer Programming*, Gerald Weinberg reminds us, “The human eye has an almost infinite capacity for not seeing what it does not want to see . . . . Programmers, if left to their own devices, will ignore the most glaring errors in their output—errors that anyone else can see in an instant (Weinberg, 1998)."

This chapter is dedicated to structured techniques for letting others look over our software development work. The techniques discussed in this chapter are called static techniques because they do not involve the execution of a program. Using these techniques, we can overcome our human shortcomings for finding our own mistakes, and we can brainstorm alternative approaches we wouldn’t think of on our own. The first set of techniques for gaining this valuable input are various forms of periodic software reviews. The second technique is a more continual practice, pair programming. Even though the second technique is called pair *programming*, this technique is used on many phases of the software development process on many types of artifacts—requirements documents, design documents, implementation code, test cases, and so on.

We use these techniques to get defects out of our work as quickly and efficiently as possible. The longer a defect remains in our work, the harder and more time consuming it is to get out (Boehm, 1981). We also use these techniques to learn from each other. Each of us has our own skills, approaches, and techniques, and we have a lot to learn from each other.
1 Software Reviews

Software reviews are used for quality assurance. Software reviews are a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code review, design review, formal qualification review, requirements review, test readiness review. (IEEE, 1990) There are several varieties of software review; they differ according to the size of the review group and the formality of the review meetings. In this section, we’ll learn about personal reviews (which are done solo by the creator of the artifact), walkthroughs (which are done informally with the artifact creator and one or two other people), and inspections (which are done formally with the artifact creator and up to four other people).

1.1 Objectives of Software Reviews

There are four explicit objectives for software reviews, as follows:

- To detect errors in program logic/structure or inconsistencies from one artifact to the next. Harlan Mills’ believes that “programming should be a public process” (Brooks, 1995). Exposing programs to others helps quality, both through the pressure by peers to do things well and because peers spot flaws and bugs (Brooks, 1995) that an individual might not.

- To make sure the intention of the artifact is clear (the more clear the better)

- To verify that the design and/or software meets its requirements

- To ensure software has been developed in a uniform manner, using agreed-upon standards

1.2 Beneficial Side Effects of Software Reviews

In addition to these explicit objectives, when reviews involve group participation, the reviews have additional beneficial side effects for the development group. First, reviews are an excellent means of learning about the overall system and about the techniques of teammates so as to improve communication within the team. Secondly, by working together several people on the team become somewhat familiar with the details of the artifact under review. This additional knowledge is helpful when the creator of the artifact is not available and the review participant must interact with the artifact. Finally, there’s a psychological benefit for the creator of the artifact. When we know that others will be looking at our documents or code, we have more incentive to make things clear and simple. As a result, our work is generally of higher quality.

2 Types of Software Reviews

Some might think that if only we didn’t make mistakes, reviews would not be necessary. But, even experienced programmers typically make about 100 defects per thousand lines of code (Humphrey, 1995)! Early discovery and removal of defects is vital so these defects do not propagate to the next step in the software process. We now describe three different types of software reviews for removing defects: personal reviews, walkthroughs, and software inspections.
2.1 Personal Reviews
On one end of the software review spectrum are personal reviews—here you privately review your own work. Think back to a time when you’re asked someone to proof your term paper for you. You wouldn’t give your reviewer your first draft, would you? You’d go through your term paper to polish it up and find the glaring errors before showing it to someone else. The same goes with software. Before others see your work, you should examine your own products via a personal review. Humphrey (Humphrey, 1995) reminds us that someone who inspects your work is making you a gift of time solely to help you improve the quality of your product. To show your appreciation, you should treat that time as important by ensuring your code is as clean as you can before submitting it for the inspection.

A Checklist for a Personal Review
To prepare for a personal review, it is a good idea to create a checklist of questions designed to detect common errors. As you proceed through the review, ask yourself the questions on the checklist. For example, the following five sample questions could be on a checklist for a requirements document checklist:

1. Are all requirements traceable back to a specific user need?
2. Are any requirements included that are impossible to implement?
3. Could the requirements be understood and implemented by an independent group?
4. Are security requirements specified for each function?
5. Is there a glossary in which each term is defined?

Finding Defects with a Personal Review
When you do a personal review, it is best if you print out the work you will be reviewing, such as a requirements document, a design document, code, or a test plan. Then, methodically, step through your work and through the checklist, trying to identify any possible errors. The objective of a personal review is to find and fix as many defects as possible before you implement, inspect, or test the design and/or the program. Research has shown that with practice you can remove between 50%-80% of your defects by doing a thorough personal review (Humphrey, 1995). Every defect you remove from of your work on your own saves your teammates time in later inspections, testing, and field support and improves the quality of your product.

2.2 Walkthroughs
Going up the software review spectrum one step brings us to walkthroughs. A walkthrough is a static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violations of development standards, and other problems (IEEE, 1990). At least one other person attends a walkthrough with the creator of an artifact. Generally, no preparation is done before a walkthrough, and no formal follow-up is done after a walkthrough. This form of software generally follows a presentation format. The developer first makes an overview presentation of the software element(s) under review.
Then, he or she traces the design or code step by step. (IEEE, 1988) The developer also gives a detailed description of how the program handles a typical application. The audience, which can include customers/users and other team members, raises issues and asks questions. Errors, suggested changes, and improvements are noted as the walkthrough progresses. A walkthrough can therefore be effective at discovering omissions, and resolving misunderstandings; it can also be used to educate users or team members about an application.

There are three roles for walkthroughs (IEEE, 1988). These roles are defined below:

- **Author**: The author of the material presents his or her work.
- **Moderator**: The moderator handles the administrative aspects of the walkthrough, such as determining the schedule and distributing materials, and ensures it is conducted in an orderly manner. The moderator prepares a statement of objectives for the meeting.
- **Recorder**: The recorder writes down the comments made during the walkthrough. The comments pertain to errors found, questions of style, omission, contradictions, and suggestions for improvement and alternative approaches.

### 2.3 Software Inspections

Similar to walkthroughs, software inspections involve the author creator and several other people. An inspection is a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems (IEEE, 1988). Inspections are a more formal type of software than either personal reviews or walkthroughs. The style of software inspection we will describe is often referred to as Fagan-style (Fagan, 1976) inspection, named after the software engineer who devised the practice, Michael Fagan. Software inspections generally involve three to six participants.

Organizations that include inspections in their development process generally have rules or protocols for carrying out the inspection meetings. Artifacts that will be inspected must be distributed to participants a set number of days prior to the meeting. Participants are required to review the artifact prior to the meeting so that they are prepared for an effective and efficient meeting. However, this pre-inspection preparation must not take more than two hours. Similarly, the inspection must also not last more than two hours lest the participants get too tired to provide useful input.

**Roles within an Inspection**

Additionally, a Fagan inspection requires that several participants to be present, each with a particular role to play. For smaller reviews, participants may take on more than one role. The roles (IEEE, 1988) are defined below:

- **Author**: To no surprise, the author is the person who created the document being inspected. However, as opposed to the authors role in walkthroughs, he or she is present at the inspection to answer questions to help others understand the work but does not step through the work; the reader does that. The authors listens to
the input of the inspection team but should not to “defend” his or her work. The author does not take on any of the four roles defined below.

- **Moderator:** The moderator chooses the inspection team, schedules the inspection meeting, ensures the artifact to be review are complete, and distributes the materials. In the inspection meeting, the moderator runs the inspection and enforces the protocols of the meeting. The moderator’s job is mainly one of controlling interactions and keeping the group focused on the purpose of the meeting – to discover (but not fix) deficiencies in the document. The moderator also ensures that the group does not drift off onto a tangent and that everyone sticks to a schedule.

- **Reader:** The reader leads the inspection team through the software element(s) in a logical and comprehensive fashion. He or she calls attention to each part of the document in turn – paraphrasing or reading line-by-line as appropriate. The reader paces the inspection.

- **Recorder:** Whenever any problem is uncovered in the document being inspected, the recorder describes the defect in writing. After the inspection, the recorder and moderator prepare an inspection report.

- **Inspectors:** The inspectors raise questions and suggest problems with the document. Inspectors are not supposed to “attack” the author or the document but instead they should strive to be objective and constructive. Everyone except the author can act as an inspector. Often inspectors are chosen to represent different viewpoints, for example requirements, design, code, test, project management, quality management.

**After the Inspection**

Upon completion of the inspection, the team of participants decides if the artifact (1) can proceed to the next stage with minor changes; (2) needs to be fixed and re-inspected; or (3) needs to be scrapped and done over. It is very important that none of the participants is the supervisor of any of the other participants (especially the author) and that inspection data is in no way used in employee performance evaluations.

Organizations that have embraced inspections have often found that they have far fewer test defects (Hamlet and Maybee, 2001). Despite the advantages of these inspections, unfortunately, these reviews are often not done as much as they should be. There are several reasons the reviews are not done:

- Developers simply don’t believe that the reviews are worth their time—they’ve got a deadline to meet. Instead, these same developers spend endless hours in long, error-prone debugging sessions, finding errors that could have been efficiently found in a review.
- Developers might have ego problems in reviews. They might have trouble admitting their own mistakes and don’t want a room full of people seeing their defects. However, we need to develop an egoless programming (Weinberg, 1999) culture where we each learn from each other and benefit from each others’ input so we can grow as software engineers and so we can produce higher quality products.
Some software engineers avoid inspections because they find inspections boring.

3 Pair Programming
Pair programming is a technique that can be used to complement software reviews or, sometimes, as an alternative to reviews. Pair programming is a style of programming in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test (Williams and Kessler, 2003). Pair programming has been practiced sporadically for decades (Williams and Kessler, 2003); however, the emergence of agile methodologies and Extreme Programming (Beck, 2000) has recently popularized the pair programming practice. Pair programming has been shown to have many of the benefits of reviews while also eliminating the programmer’s distaste for reviews so that at least one form of review is actually performed.

3.1 The Driver and Navigator
One of the pair, called the driver, types at the computer or writes down a design. The other partner, called the navigator, has many jobs. One of these is to observe the work of the driver—looking for tactical and strategic defects in the driver’s work. Some tactical defects might be syntax errors, typos, and calling the wrong method. Strategic defects occur when the driver is headed down the wrong path—what driver and navigator are implementing just won’t accomplish what it needs to accomplish. The navigator is the strategic, longer-range thinker of the programming pair. Because the navigator is not as deeply involved with the design, algorithm, code or test, he or she can have a more objective point of view and can better think strategically about the direction of the work.

Another benefit of pair programming is that the driver and the navigator can brainstorm at any time the situation calls for it. An effective pair programming relationship is very active. In an effective pairing relationship, the driver and the navigator continually communicate. Periodically, it’s also very important to switch roles between the driver and the navigator.

3.2 Pairing during All Phases of Development
The name of the technique, pair programming, can lead people to incorrectly assume that you should only pair during code development. However, pairing can occur during all phases of the development process, in pair design, pair debugging, pair testing, and so on. Programmers could pair up at any time during development, in particular when they are working on something that is complex. The more complex the task, the greater the need for two brains.

3.3 Why Pair Program?
Some people think that having two people sit down to develop one artifact must be a big waste of resources. Managers are especially concerned about this since they think they
will have to pay two programmers to do the work one could do. Even students are concerned about this because they think they might have to spend twice as long on their homework. However, some research results show that these concerns do not materialize.

**Higher-Quality Code**

Previous research with senior-level undergraduate students at the University of Utah showed that pairs developed higher quality code faster with only a minimal increase in total time spent in coding. For example, if one student finished a project in ten hours, the pair might work on it for five and a half hours (for eleven total hours of time between the two). The code produced by the pairs in the study also passed 15% more of the automated test cases, demonstrating that the pairs produced code of higher quality. (Williams, Kessler et al., 2000; Williams, 2000)

At North Carolina State University, student pair programmers in beginning computer science classes generally performed better on projects and exams and were more likely to complete the class with a grade of C or better than did their solo counterparts. Results also indicate that pair programming creates a laboratory environment conducive to more advanced, active learning than traditional labs; students and lab instructors report labs to be more productive and less frustrating. (Williams, Wiebe et al., 2002; Williams, Yang et al., 2002; Nagappan, Williams et al., 2003)

**Enhanced Morale, Teamwork, and Learning**

Pair programming offers additional benefits, including the following:

1. **Increased Morale.** Pair programmers are happier programmers. Several surveys were taken of pair programmers in the North Carolina study discussed above. Ninety-two percent of them indicated that they enjoyed programming more when they worked with a partner. Ninety-six percent of them indicated they felt more confident in their product when they worked with a partner. (Williams, 2000)

2. **Increased Teamwork.** Pair programmers get to know their classmates much better because they work so closely together. (Cockburn and Williams, 2001) It makes school more enjoyable when you can walk into a classroom or a lab and really know several of the people in the class. Classmates then seem more “approachable” when you have a question about the class.

3. **Enhanced learning.** Pairs continously learn by watching how their partners approach a task, how they use their language capabilities, and how they use the development tools. (Cockburn and Williams, 2001)

**3.4 How Does Pair Programming Work?**

It may seem odd that two people can sit down at one computer and finish in about half the time, with higher quality code, and enhanced morale, teamwork, and learning. But studies have shown that pairing makes us work differently. As was done in (Williams
and Kessler, 2003), we will discuss six “hows” and “whys” that contribute to the great results of pair programming. They are pair pressure, pair negotiation and brainstorming, pair courage, pair reviews, pair debugging, and pair learning.

1. **Pair Pressure**

Pair programmers put a positive form of pressure on each other. This pressure causes the partners to react in several ways.

- **Programmers say they work harder and smarter on programs because they do not want to let their partner down.**

  When you’re working alone, you can (intentionally or unintentionally) stare at the computer screen and/or waste time on mindless tasks. Simply the presence of another can draw us out of our tired or disgruntled mood and cause us to get to work. We just don’t want to let the person next to us down. Or, we’re embarrassed to disappoint our partner or to look like a slacker.

- **Programmers say they work very intensively because they are highly motivated to complete the task at hand during the session.**

  “Two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don’t check e-mail messages or favorite Web pages; they don’t waste each other’s time (Wiki, 1999).” When you work alone, you can meander through the day and surf the web when you want. When you pair, you need to coordinate schedules for getting together outside of class. During these precious prearranged hours, you’re motivated to work intensively to get to the ‘goal line’ before you need to go your separate ways. Pairing provides explicit deadlines that motivate you to finish the task and not let it extend into another day.

2. **Pair Negotiation and Brainstorming**

The second of the pair programming behaviors we discuss is one we call *pair-negotiation*. We use the term pair negotiation to describe how two pair programmers arrive at the best solution together. When pairing is working at its best:

Each person brings to the partnership his or her own set of skills, abilities, and outlooks. Both share the same goal for completing the task. Each person has a suggested alternative for attacking a joint problem. The partners must negotiate how to jointly approach the problem. In this negotiation, they evaluate more alternatives than either one would have considered alone, whereas, a person working alone tends to pursue the first approach that comes to mind. Together, the partners consider and include each other’s suggestions and determine the best plan of attack.

Another approach similar to to pair negotiation, is that of *collaborative teams*. These participants consistently report that together they can develop solutions to seemingly impossible problems. *Pair brainstorming* describes the give-and-take process in which two people work together to resolve a problem. They share their knowledge and energy,
chipping steadily away at the problem, evolving a solution to the problem. An effective technique of brainstorming is to build upon the ideas of others. A subtle difference in pair brainstorming though, is that the driver might actually be working out a design or implementing part of the problem, realizing that he or she may ultimately come to a dead end in their problem resolution. The navigator, while watching the driver’s partial design or implementation, also begins thinking about the next step. When the driver hits the dead end, the navigator is often prepared to take over and lead the way. Often, the cycle continues until the problem is resolved.

3. Pair Courage

Having a partner is a tremendous courage builder. You can say to each other, “Does this look right to you?” Getting affirmation from our partner gives us the confidence to do things we might be afraid to do alone or that we might simply avoid and never do.

Teachers see that if they give students a problem, ask them to solve it during class, and ask them to volunteer the answer for the class, they usually get a room full of blank stares and the same two or three hands up. Most people will avoid sharing their answer because they don’t want to look stupid if they are wrong. However, if the teacher tells the students to work out the problem with their neighbor and then asks for responses, many pairs will willingly share their answer. When we work with someone else, we can piece together enough knowledge to feel confident in what we’re doing. And, if it looks right to me and it looks right to you – guess what? It’s probably right! Strength in numbers!

Working with a partner also gives us courage to admit when we don’t know something. Developers by themselves tend to be embarrassed when they don’t know something and will try to muddle through on their own rather than ask for help from their peers. When two people don't know something, there is a joint realization that it is time to seek help.

4. Pair Reviews

With pair programming, problem identification occurs on a minute-by-minute basis. With pair programming, “four eyeballs are better than two,” so that numerous defects are prevented, removed right from the start.

5. Pair Debugging

Every person has experienced problems that can be resolved simply through the act of explaining the problems to another person.

... [an] effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind; I see what's wrong. Sorry to bother you." This works remarkably well; you can even use nonprogrammers as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them
to the teddy bear before they could speak to a human counselor. (Kernighan and Pike, 1999)

Others have cited the need just to have a cardboard cutout of their favorite guru or a rubber duck. When they have a problem, they must explain the problem to the inanimate helper, often finding the elusive flaw in the logic. Doubtlessly, it is better to have an animate person to debug with than an inanimate helper. A person will ask questions and will likely cause you to frequently explain your reasoning. The questions and answers can lead to great revelations.

6. Pair Learning
Knowledge is constantly being passed between partners, from tool usage tips to programming language rules to design and programming techniques. The partners take turns being the teacher and the student on a minute-by-minute basis. Even unspoken skills and habits cross partners (Cockburn and Williams, 2000).

3.5 A Fly on the Wall
What’s it like to participate in pair programming? To help you envision what it’s like, let’s be a fly on the wall one morning when a class of students is getting started with a new lab. When students enter the room that morning, they are told to go to a web site to find out who their partner is. As students file in, partners find each other and sit down together. This morning, we’ll watch Christopher and Kimberly work on their programming project. As you’ll see, Kimberly has more experience programming than Christopher does. (Often the skill levels of the pairs is not the same.) Kimberly will learn the programming techniques even better by explaining to Christopher what they are working on. Christopher will help Kimberly because he’ll know something she doesn’t know and he’ll catch some of her mistakes. Those “little” mistakes can often cause long debugging sessions—so it’s best for both of them if these mistakes are fixed before the program is ever compiled.

The project this week is on control structures (you know – if, if/then/else, for loops, do/while). Wait . . . they’re getting started . . .

Christopher: Hi, Kimberly. I guess we’ll be working on this project together. Do you live on campus?

Kimberly: Yeah. I live in Sullivan. Do you?

Christopher: I do too – in a triad in the Gold dorm, with a couple of guys I knew in high school. Did you read the program last night? It’s a credit card application program. It doesn’t look too hard. But, I’ve never used control structures before and that’s what this lab is about. Have you used them before?
Kimberly: Yeah, I took a programming class in high school, but that was in C++. I would guess it’s pretty similar in Java. Why don’t I drive first and you can drive in a half an hour, OK? Since I’m not exactly sure of the syntax, why don’t you open your book to Chapter 3. Then, if we need to look something up, you can be ready.

Christopher: OK. Good idea. I’ll also get my notebook out so I can look at the class notes too. Alright, let’s get started. The program is about processing credit card applications. So, I’m going to take a stab in the dark \{grin\} and guess we will have a credit card class.

Kimberly (laughing): Oh, you are a rocket scientist, aren’t you? Let’s draw out some quick pseudocode for the first requirement. What is it – OK, I see . . . there’s a requirement to calculate the credit limit for college students. It says that freshman and any students under the age of 18 can’t get a credit card. Sophomores and juniors that have a GPA of under 3.0 can have a credit limit of $1000. Everyone else can get $2000. OK, I’ll write down some pseudocode. Tell me if you see anything wrong, OK?

Christopher: Got it.

Kimberly (writing):

```
Class – creditCard
Method – StudentCreditLimit

    if (under 18 or freshman) limit = 0
    if (sophomore or junior and GPA under 3.0) limit = 1000
    else limit = 2000
```

How’s that?

Christopher: Looks good. Maybe instead you could set limit to 2000 to start – as the default – and then change the value of limit if either of those two conditions is met. What do you think?

Kimberly: I like that! That will make the logic a bit simpler. \(\text{(cuts and pastes the last line to the first line – deleting the else)}\). OK. So, what do you think we need to pass as parameters to the method?

Christopher: I guess we need to pass the year, like freshman, sophomore, junior, senior, the GPA, and the age.

Kimberly: Exactly! What kind of data types do each of them need to be?

Christopher: Hmm . . . I’d suggest we pass the year as a String, the GPA as a double – cause it has a decimal value, and age as an integer.
Kimberly: That’s good. OK, let’s get started.

(typing)

```java
public class CreditCard {

    private int limit;
    public CreditCard() {
    }
    public int creditLimit(String year, double GPA, int age) {
    }
}
```

How does that look so far, Christopher?

Christopher: Looks good. That’s pretty much what we discussed. Oh yeah, but you returned an int from the CreditLimit method. I guess that’s the value for the credit limit that we’ll be computing. We’ll have the method return that value, right? Oh, yeah – and you have a private attribute limit. Got it. OK. So let’s fill in the CreditLimit. You should start by setting the limit to 2000, the default.

Kimberly (typing):
```java
public int creditLimit(String year, double GPA, int age) {
    limit = 2000;
    return limit;
}
```

OK. So that’s the most basic function. We set limit to the default and return that value. That look OK? Let’s compile and see if it works so far. {pause} OK. So far so good. What now? Let’s test for the case when the person’s a freshman or under 18.

(typing after the first line in the method started above)
```java
    if (year.equals("FR") || age <= 18)
        limit = 0;
```

Christopher: No, wait. It should just be age < 18, not <= 18. They can’t have a credit limit if they are under 18. They can if they are 18.

Kimberly: Oh, good catch. I definitely would have messed that up. {deletes the =} OK, now for the homestretch . . .

```java
    else {
        if ((year.equals("SO") || year.equals("JR")) && (GPA < 3.0))
```
Christopher: Looks great to me. Let’s test, OK?

Kimberly: Sounds good!

Together, Christopher and Kimberly got the job done. They brainstormed and, in the end, got the job done well. In the process, they got to know each other.

### 3.6 Distributed Pair Programming

So far, the kind of pair programming we have been discussing requires that the two programmers sit right next to each other. This may not always be possible. In industry, increasingly team members are in separate locations, that is, team members are practicing distributed pair programming. For example, students may prefer to work from their dorm rooms, rather than going to the lab to work with their partners. And students enrolled in distance education courses may not ever be able to meet each other face-to-face. Some studies of distributed pair programming have been done with students at both North Carolina State University and the University of North Carolina —Chapel Hill. These studies seem to show that pairing over the Internet shows a great deal of potential when compared with distributed non-paired teams in which each student works alone and code is integrated later. (Baheti, Williams et al., 2002) In these studies, the students used desktop sharing software and headsets/microphones to communicate.

### 4 The Economics of Quality Assurance

How can we justify the time spent on pair programming and software reviews? It might seem faster to skip these steps and move right into software test.

#### 4.1 General Research Findings

Research studies (including (Fagan, 1986; Gilb and Graham, 1993; Basili and Selby, December 1987)) have been done to assess whether the time invested in software reviews is worthwhile. Researchers have found that reviewing was more effective and less expensive than testing in discovering program faults and that more than 60 % of the errors in a program can be detected using informal program inspection. As said above when pairs work, they produce higher quality code.

#### 4.2 Ease of Finding and Fixing Defects

In both reviews and pair programming, you find the defects directly and you deal with the problems that are identified. (In reviews, you should only identify problems, but not try to solve the problems on the spot. With pair programming, you solve problems on the spot.) By this, we mean that in a review, a fellow programmer could tell you, “On line 20, you put limit \(\leq 100\), but that should be \(< 100\).” You say, “Yes, you’re right!” you correct your paper and fix it in your code later.
In black-box testing, however, you get only symptoms. For example, you run a planned test case and you get the wrong answer. The wrong answer is just a symptom of the problem in your code. Once you find that symptom, you must figure out why in the world you got the wrong answer – which line of code would cause such a symptom? The time you spend tracking down the exact problem that caused the improper behavior is called debugging. Depending upon the size of your program, this debugging time could be very time consuming. Most often there is no relationship between the size of the defect you find and how long it takes to find the defect (that <= sign in the example above could take hours or days to find.).

You should feel highly motivated to find and fix as many defects as possible before you head into your black-box testing phases. Consider that the review/pair programming form of defect identification and correction is fairly efficient, is quite predictable, and has been shown to be able to remove more than half of the defects in your project. You know that you and your four reviewers will sit together for two hours and you will find many problems. You know that you and your pairing partner will spend several hours together to write high quality code. However, once you start doing black-box testing, you enter the chaos zone. At this point, even the smallest defect could take many hours to find and fix —and these hours unpredictable and are often very frustrating. The relative ease of tracking down and fixing problems with reviews and pair programming when compared with the difficulties of tracking down and fixing symptoms with debugging sessions is why these practices are more efficient.

4.3 Garbage In, Garbage Out (GIGO)

As good as these reviews and pair programming are, any quality assurance practice (such as pair programming, reviews, and testing) cannot remove all the defects in a document or code. These practices are only imperfect filters that can remove a percentage of your defects. The percentage of defects removed by a quality assurance activity is called the yield of the practice. For example, if an inspection actually gets out 40% of the defects in the code, the inspection is said to have a 40% yield.

Because all of the quality assurance practices are imperfect filters, the more defects that are in the program, the more defects will escape to your customer. (Hence, the title of this section – Garbage In, Garbage Out.) Also, the more filters you have, the more defects you can remove. Consider three hypothetical programs (as shown in Figure 1) that would start with 100 errors in each of them (however, pair programming prevents half of these from being injected into the program at the coding stage). For the first program, the development team works solo and does inspection. For the second program, the development team also works solo but does not do inspection. Finally, the third program was developed by programmers working in pairs without inspection.

4.4 Economic Analysis of Reviews

Quality is certainly one concern we have in our development. However, if obtaining this degree of quality is exceedingly expensive, we may not be able to afford these quality
assurance steps. We must analyze the economic feasibility of these steps. The previous discussion and Figure 1 show how having additional quality assurance activities can reduce the number of defects that are delivered to a customer. The top diagram in Figure 1 shows the case of solo programming with inspection. The middle diagram shows the case of no inspection. The bottom figure illustrates pair programming. Even more defects can be prevented from being “delivered” to a customer if both pair programming and reviews were used.

**Figure 1: Quality Assurance Filters**

![Diagram of Quality Assurance Filters]

**Solo programming with inspection**

100 errors → Inspection (50%) → Testing (50%) → 25 errors → To customer

**Solo programming without inspection or pair programming**

100 errors → Testing (50%) → 50 errors → To customer

**Paired team without inspection**

Pair Programming → Pair Programming → Testing (50%) → 25 errors → To customer

We consider two simple scenarios to explain the economics of quality assurance. The example makes two important but realistic assumptions.

- Experienced software engineers normally inject about 100 defects/KLOC (KLOC = thousand lines of code). About half of these defects are found by the compiler (the compiler has 50% yield).
- In industry, defects that escape from the compiler take on average eight hours each to find and fix in the testing phase. Eight hours/defect may sound like a lot,
but it is realistic and actually quite low. It can be hard to find the defect. Sometimes a software engineer will have to get in their car or jump on a plane to go to a customer site to help find and fix the defect. All this time adds up!

**Scenario One: Solo Programming, No Inspection**

Consider the case of solo programming without inspection (middle diagram of Figure 1). For a 50 KLOC program, there will be \(50 \times 100 = 5,000\) defects. Half of these will be caught by the compiler and the rest (2,500) will escape to the testing phase, since the solo programmers do not perform inspection.

No practice is a perfect filter (it won’t get out all the defects that are in the project.) Testing will generally identify only half of the defects that enter the phase. How many programmer hours will it take to find and fix this half of the defects?

\[
(2,500 \text{ defects})(0.5 \text{ test yield}) = 1,250 \text{ defects removed in test} \\
(1,250 \text{ defects})(8 \text{ hours/defect}) = 10,000 \text{ hours to find and fix those defects}
\]

- How many weeks would it take 10 people to do this work, assuming 40 hours work/week each?

\[
(10,000 \text{ programmer hours})/(10 \text{ people})(40 \text{ hours/week}) = 25 \text{ weeks (or almost half a year)!}
\]

- How many defects will escape to the customer?

\[
2,500 - 1,250 = 1,250 \text{ defects escape to the customer}
\]

- What would the impact be to the quality of the product if management said “You have three months (12 weeks) to test this product.”

Unfortunately, this is what often happens. The testing phase comes right before the product is released. Often testers must compromise on the time so that the product can be released on schedule. At 8 hours/defect, each person can only remove 5 defects per 40 hour week. In 12 weeks, this team could only remove:

\[
(12 \text{ weeks})(10 \text{ people})(5 \text{ defects/week}) = 600 \text{ defects}
\]

This means that 2,500 – 600 defects = 1,900 defects would escape to the customer.
Scenario Two: Solo Programming with Inspection or Pair Programming without Inspection

These cases are shown in the top and bottom diagrams of Figure 1. For the sake of this example, we will assume that pair programmers produce code of equal quality to reviewed code, though there are no research results to back up this claim. We do know of several industrial organizations that are beginning to offer employees an alterative to either formally inspect their code or to pair program; their anecdotes support that this alternative provides similar benefit. However, there are no research results that conclusively show that these alternatives are equal.

We again use the same assumptions as above (50 KLOC, 100 defects/KLOC, 50% yield from the compiler, 8 hours to remove each defect in test).

For a 50 KLOC program, there will be (50)(100 defects/KLOC) = 5,000 defects. For the solo programming group, half of these will be caught by the compiler and the rest (2,500) will escape to the inspection phase.

Inspections will generally identify only half of the defects that enter the phase. In industry, approximately 0.5 hours are spent to find and fix each defect in an inspection phase. (Defects might be found quite rapidly in an inspection. However, we must consider that four people might be attending the review. If 8 defects are found in an hour-long review, the average time is calculated 4 person hours/8 defects or 0.5 hours/defect).

How many programmer hours will it take to find and fix this half of the defects in the inspection?

\[(2,500 \text{ defects}) \times (0.5 \text{ inspection yield}) = 1,250 \text{ defects removed in inspection} \]
\[(1,250 \text{ defects}) \times (0.5 \text{ hours/defect}) = 625 \text{ hours to find and fix those defects} \]

Now, 1,250 defects remain in the product as the product enters the test phase from either the solo/reviewed group or the pair programming group. We assume the pair programming group had not done an inspection, but instead had a continuous review as they worked. The test phase will still again only find half of the remaining defect.

How long will the test phase need to be?

\[(1,250 \text{ defects}) \times (0.5 \text{ test yield}) = 625 \text{ defects removed in test} \]
\[(625 \text{ defects}) \times (8 \text{ hours/defect}) = 5,000 \text{ hours to find and fix those defects} \]

- How many weeks would it take 10 people to do this work, assuming 40 hours work/week each? (Would they be having fun?)
(5,000 programmer hours)/(10 people)(40 hours/week) = 12.5 weeks

- How many defects will escape to the customer?

1,250 – 625 = 625 defects escape to the customer

- What would the impact be to the quality of the product if management said “You have three months (12 weeks) to test this product.”

Now, this deadline would not be a problem – the team would just need to work minimal overtime to complete test.

In summary, the solo/no review group needed to work 10,000 programmer hours in testing. The solo/review group needed to work 5,000 programmer hours in testing plus they had to dedicate 625 hours to inspect for 5,625 hours of quality assurance activity. The solo/review group saved 4,375 hours when compared with the solo/no review group and had higher quality code. We will also assume the pair programming group had similar results to the solo/review group – minimal increase in programmer hours due to doubling up, 5,000 hours in test, and high quality code.

To re-emphasize, often rushed development teams decide not to do any reviews and/or decide not to pair program because they “don’t have the time.” Remember the scenarios above when you are tempted to do the same. Finding, fixing, and preventing defects as early and efficiently as possible should be your goal.

5  Recommendations for Effective Reviews and Pair Programming

Reviews and pair programming can either be highly effective or a huge waste of resource, depending upon whether protocol is followed and whether the people involved contribute and are receptive. Below we give some recommendations for you to make your reviews and your pairing as effective as possible. We suggest that you take these recommendations and make them habits.

5.1  Recommendation 1: Practice Humility

“Ego-less programming,” an idea surfaced by Gerald Weinberg in The Psychology of Computer Programming (Weinberg, 1998) a quarter of a century ago, is essential for effective pair programming. Excessive ego can manifest itself in two ways, both damaging the collaborative relationship and the spirit of a review. First, having a “my way or the highway” attitude can prevent the programmer from considering other’s ideas. Secondly, excessive ego can cause a programmer to be defensive when receiving criticism or to take this criticism as an expression of mistrust. However, all must remember to put the team’s progress above his or her own ego.
None of us, no matter how skilled, is infallible; all of us, no matter how skilled, can benefit from the input of another. John von Neumann, the great mathematician and creator of the von Neumann computer architecture, recognized his own inadequacies and continuously asked others to review his work.

And indeed, there can be no doubt of von Neumann's genius. His very ability to realize his human limitation put him head and shoulders above the average programmer today . . . . Average people can be trained to accept their humanity -- their inability to function like a machine—and to value it and work with others so as to keep it under the kind of control needed if programming is to be successful. (Weinberg, 1998)

Weinberg also shares (Weinberg, 1998) a true story about a programmer seeking review of the code he produced. On this particular “bad programming” day, this individual ego-lessly laughed because his reviewer found 17 bugs in 13 statements. However, after fixing these defects, this code performed flawlessly during test and in production. Think how much worse the programmer's life would have been if he'd been too proud to accept the input of others or had viewed this input as an indication of his inadequacies.

5.2 Recommendation 2: Be Tactful

The previous recommendation encouraged you to know how to take objective criticism and suggestions. This recommendation encourages the opposite – know how to give objective criticism. In both pairing and in reviews, remember never to infer (or blatantly say) that your partner or the author is inferior to yourself or has made a stupid mistake. Remember, to err is human, and we are all human and are always learning, particularly in the dynamic field of software development. Eventually (in a few minutes or a few days), you will be the driver or the subject of an inspection. At all times, treat others as you would like to be treated.

5.3 Recommendation 3: Respect the Protocol

Software reviews have protocols. For example, personal reviews rely on checklists, which need to be developed. In walkthroughs, authors present their designs and code in an informal way. In software inspections, there are specified roles for the participants, and all participants should come to the inspection prepared. Defects that are identified must be recorded.

Pair programming also has its protocols. The partners assume roles of driver and navigator. However, they swap these roles periodically. Pairs converse almost continuously.

By using these protocols, which have been gradually established by those who have used these techniques before you, you can maximize the effectiveness of these development practices.
5.4 **Recommendation 4: Know When to Stop**

“Pair programming is exhausting but productive. (Wake, 2001)” Because pair programmers do keep each other continuously focused and on-task, it can be a very intense and mentally exhausting. Periodically taking a break is important for maintaining the stamina you need for another round of productive pair programming. During the break, it is best to disconnect from the task at hand so that you can approach it with freshness when restarting.

Reviews are also exhausting. Through many experiments it has been established that reviews should be no longer than two hours. If a review lasts longer than this, the participants start to get exhausted and the productivity of the review drops significantly. Over time, you will learn how much you can review in this period of time and will schedule the reviews accordingly.

6. **Summary**

A main goal of software reviews and pair programming is to remove defects in software products. As has been discussed, both of these techniques also provide excellent learning environments for team members. Additionally, pair programming also prevents defects from being injected in the product in the first place. These ideas are summarized in Table 1.

<table>
<thead>
<tr>
<th><strong>Table 1: Key Ideas for Software Reviews and Inspection</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Key Idea" /></td>
</tr>
<tr>
<td>The sooner a defect is found and fixed in a product, the less expensive it is for the product – and the less frustrating it is for the software engineer.</td>
</tr>
<tr>
<td>Software engineers should review their code via a personal review to remove as many defects as possible before others get involved with helping them remove their defects.</td>
</tr>
<tr>
<td><img src="image" alt="Key Idea" /></td>
</tr>
<tr>
<td>Inspections are more formal reviews of software artifacts that involve three to six people.</td>
</tr>
<tr>
<td><img src="image" alt="Key Idea" /></td>
</tr>
<tr>
<td>Not quality assurance filter (such as reviews, pair programming, and testing) removes all the defects in a product.</td>
</tr>
</tbody>
</table>

© Laurie Williams 2004
they are delivered to a customer. Once a customer gets a product with a defect, the defects make the customer less delighted with the product, the defects are much more costly to find and fix.

**Glossary of Chapter Terms**

<table>
<thead>
<tr>
<th>Word</th>
<th>Definition</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>inspection</td>
<td>A static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems.</td>
<td>(IEEE, 1990)</td>
</tr>
<tr>
<td>pair programming</td>
<td>A style of programming in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test.</td>
<td>(Williams and Kessler, 2003)</td>
</tr>
<tr>
<td>review</td>
<td>A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include code review, design review, formal qualification review, requirements review, test readiness review.</td>
<td>(IEEE, 1990)</td>
</tr>
<tr>
<td>walkthrough</td>
<td>A static analysis technique in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code, and the participants ask questions and make comments about possible errors, violations of development standards, and other problems.</td>
<td>(IEEE, 1990)</td>
</tr>
</tbody>
</table>

**References**


