Extracted from:

Ship It!

A Practical Guide to Successful Software Projects

Pair Programming

Pair programming puts two team members at one computer. One types the code while the other tries to pull back and keep an eye on the big picture. One works in the details of the code and the language syntax while the other is trying to decide if a particular algorithm is the right one to use to solve a problem. The second person spots problems such as coding errors, spelling problems, and bad variable names. From time to time the developers switch roles.

Some people love the practice while others never seem to acclimate to it. We've found it to be a powerful practice when used appropriately (with the right people). For a more in-depth look at this intriguing practice, visit the aptly named web site http://www.pairprogramming.com/.

Review All Code

Small, frequent code reviews keep your code clean, simple, and tidy. You can avoid the traditionally unpleasant code reviews that involve dozens of developers and require days of preparation (a.k.a. *The Mighty Awful and Dreaded Code Review*, hereafter referred to as *MAD reviews* for your reading enjoyment). We've found code reviews can be painless when you adhere to the following rules:

- Only review a small amount of code.
- There are one or two reviewers at most.
- Review very frequently, often several times a day.

Your goal is to move toward a habit of more frequent code reviews while not incurring the potential culture shock (or the perceived overhead) of *pair programming*. Many environments just aren't ready for that level of interaction; the cost of breath mints alone could break a thriving company! So instead, strive to review your code more often and in smaller chunks.

pair programming

If a week goes by without a code review, you've allowed a lot of time for serious problems to creep into your code. You likely need an outside perspective if you have been working on a tough problem that long. It's not even that another person knows better; just the act of explaining

the problem is often sufficient for you to then solve it (*The Pragmatic Programmer* calls this *rubber ducking*).[7] Waiting days before getting a code review (even if it's just an interim checkup) will probably be a long and painful experience. . . a MAD review!

To avoid MAD reviews, separate your work into the smallest possible pieces and get each one reviewed independently and committed into the source code repository. Then if there's a problem with any one area, it's easily isolated.

Programmers can easily get so caught up in the details of a particular task that they miss obvious big-picture improvements. When you stop to explain your direction and code to another person, you have to break that flow, and you'll often get a valuable fresh perspective. Sometimes we are so busy creating a road in the woods that we don't realize we are in the wrong forest, headed in the wrong direction!

Another benefit of segmenting code is that reviewers can more readily understand the code if it's divided into smaller pieces. In a fast-paced development shop, you may need to have your code reviewed many times in one day. A good rule of thumb, however, is to never work for more than two days without getting a code review. Think of code reviews like breathing. Sure, you can go a few minutes without breathing, but who wants to?

Ideally there will be one review for each feature you add (or for each bug you fix). Holding your code until you have seven new features added and fourteen bugs fixed is a recipe for the dreaded MAD review (not to mention a long, drawn-out, and unfocused effort).

If you're in the position of rewriting an intricate part of your product and you can't divide the task into smaller parts, pick one reviewer and have that person frequently do on-the-fly interim code reviews.

You'll write better code when you know someone else will see it and hold you accountable for it. This issue isn't unique to developers; it's simply human nature. Code reviews ensure that at least one other person will look at your work. You'll know that you can't take shortcuts in your code with this accountability in place.

[7] So called because the other person doesn't need to contribute anything to the conversation except an appropriate nod now and then. If you can't find a person, even a rubber duck will do.

Plenty of research shows the effectiveness of code reviews at detecting defects (bugs) in code. In fact, it's the number-one technique for finding bugs. There is none better. If you haven't done code reviews consistently, you may be in for a surprise at what you'll find.

We've actually seen variable names like mrHashy (Mister Hashy) for a hash table. However, after a single code review, that developer began to use more relevant variable names to avoid being teased by co-workers. Peer pressure can be painful yet effective.

Rubber ducking (explained previously) is a very effective way of finding and solving problems. By describing your code to someone, you'll suddenly realize things you forgot, recognize logic that just won't work, or find conflicts with some other area of the system. We want you to "talk" to the duck every time you check in code.

Besides the value in rubber ducking, other developers *will* spot bugs in your code. Having a fresh, second set of eyes to look over your code will often catch issues that never even occurred to you. You'll be getting a completely different point of view. Finding bugs in the development shop is always cheaper than finding them when the code is in the field. The return on this small investment is immense.

Code reviews are great for fostering knowledge sharing among team members. After collaborating on the review, your reviewer will have at least a conceptual idea of what your code does and you hope a detailed understanding of it. This has enormous mentoring benefits and helps in code maintenance as well.

Reviews provide a perfect opportunity for experienced developers to pass along code style and design techniques to less experienced programmers. Beyond the trivial technicalities (such as where the brackets go), code reviews give the veterans a chance to advise the greener developers on why one data structure might be better for this situation, or to point out that a pattern is emerging. Quite often, a reviewer will spot repeating code or functionality in these sessions, which can be moved to common base classes or utility classes. Your code becomes refactored *before* it gets checked into the source code management system.

Reviews also facilitate cross-training on small areas of coding details as well as big-picture concepts. Beyond "coding style," you are learning to "code with style."

Here are a few guidelines to assist you with code reviews.

Patterns

A *pattern* refers to the practice of documenting and naming common problems (and their solutions) that occur in real-world projects. There are several reasons to be a student of patterns. One is to give developers a common vocabulary. After developers have worked together, they develop a common set of terms that lets them communicate quickly and unambiguously. Patterns can jump-start that process and allow you to communicate clearly with someone you've just met (provided they are also familiar with the same patterns).

Another good reason to study patterns is to help you solve problems you haven't seen before. By reading and discussing various patterns, you learn how to solve many common problems. The question isn't whether you'll encounter most patterns but whether you'll recognize them when they cross your path.* Will you know how to cleanly solve the problem represented by the pattern, or will you stumble through several code iterations before you find an acceptable solution?

Design Patterns: Elements of Reusable Object-Oriented Software by Eric Gamma, Richard Helms, Ralph Johnson, and John Vlissides (a.k.a. the Gang of Four) is a great place to start.

The Humble Programmer by Edsger W. Dijkstra is a classic look at the state of computer science (including patterns) that is still very applicable today. The article was written in 1972! (See http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF.)

Code reviews must involve at least one other developer. In practice, it will almost always be just one other developer unless you are creating something interesting or clever that other team members want to learn about. Then feel free to include more developers. Don't go overboard, though (no more than three to four, tops); too many developers bog down the review.

Do not make code publicly available without a review. Don't add your changes to the source code from which your product is built until a review has been done. Part of the comments you include with the code's check-in should list your reviewer's name. Then, if there are questions about the reason for the code change and you're not around, there is a second person who should be able to explain it (at least at a basic level).

Refactoring

We couldn't improve on Martin Fowler's own description:

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a refactoring) does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring."*

*From http://www.refactoring.com/

Never use this code review rule as an excuse to not check in your code. If your company has a source code system that holds only production code, then put your code into a private area until it's ready. This private area might be a separate code repository or another installation of the source code management system. Use whatever tools you need to get the code off your machine so that when your machine dies (as machines are prone to do), your code doesn't die with it.

Reviewers maintain the right to reject code that they find unacceptable. When you review someone's code and it isn't commented correctly, the algorithms are not efficient, or, for whatever reason, don't be afraid to ask for revisions (but don't be picky—remember there are usually many acceptable ways to achieve the same result). As a reviewer, your job is to improve the code, not to rubber-stamp it. As Eric S. Raymond says, "Many eyes make all bugs shallow."

If your code can't be explained at a level the reviewer can understand, then the code must be simplified. As a reviewer, don't sign off on anything you don't understand and feel comfortable with. After all, your name is associated with this code as the reviewer. You are, as *The Pragmatic Programmer* says, "signing your work." Be sure it's worthy of bearing your signature.

Any code changes that are made can't break any existing automated tests. (You *do* have tests, right? See Practice 7, *Use a Testing Har-*

ness, on page 42.) Don't waste your co-worker's time asking for a code review if you haven't yet run the tests. If you require existing tests to be updated, make the changes to those tests a part of the coding before the review. Any *new* tests that you are adding should be a part of the review as well. As a reviewer, always reject code changes if you think more tests are necessary.

"First, do no harm"[8] is not so much just a code review rule as it is a rule to live by in general. Code changes are never allowed to break the product. Of course, this rule is a moot point with a good test suite in place, but there's rarely an excuse to break existing functionality. Instead of breaking an existing API, add a second API that has the extra argument (or whatever) you need.

For example, if an existing function call has to be changed, establish a schedule that maps out the elimination of the existing routine. Don't just remove the routine out from under your customers (your fellow teammates or other teams within your organization); make a conscious decision whether to keep the old routine or not. The scheduled removal is also important—don't have deprecated routines that live on for years (you know who you are!).

Rotate the reviewers you use, but don't be religious about it. Occasionally having the same reviewer back to back is fine, but avoid a "buddy system" where you always review Kevin's code, and vice versa. Also, never have a single, designated (and overworked) reviewer who your entire team goes to. Both situations defeat the cross-pollination effect you're trying to encourage.

Keep code reviews informal. Rather than schedule a meeting, just drop in on a team member and ask if it's a good time for a review. Sometimes you can review the code while it's still in your editor. Sometimes you'll print out the code diffs and carry them with you. The format or venue isn't important—just get the review.

When you introduce the code review process, you may need to appoint a few senior team members to be the mandatory reviewers; one of the senior team members must participate in every review at first. You shouldn't need them to continue in this role for more than a few months. Once your team members learn the basics, the whole team will be capable of sharing the responsibility. As the proverb says, "As

[8] Hippocratic Oath

iron sharpens iron, so one man sharpens another."[9] The point is for the team members to work together and so improve each other. Involve your team members in the sharpening process as quickly as possible.

We worked in one shop that really illustrates how code reviews can be used to leverage your senior members. We had three very senior developers and five who were decidedly not—they weren't rank novices, but sometimes they had peculiar ideas of how to fix a problem. In order to protect the product and to bring the junior developers up to the next level, all code reviews involved one of the senior team members. This let the more experienced team members instruct and teach while catching problems before they were introduced into the product. It also made the senior team members aware of misunderstandings and real issues that the junior developers faced.

These reviews were a great help to the team. We frequently spotted repeated code and summarily pulled it out and moved it into utility classes. Reviewers caught and removed code that had nothing to do with assigned work (otherwise known as *freelance refactoring*) and rejected uncommented code outright. As the team moved forward, an imperceptible (but very important) change took place.

Each of the junior team members started picking up good habits, one code review at a time. They started cleaning up code before the reviews, adding meaningful variable names, comments, and such before they were asked. Long, cumbersome routines became short and manageable.

Even better, the lessons taught in the code reviews stuck. After about three months, we changed the code review policy so that any team member could do the reviews.

If one or two of your developers routinely miss things in reviews, you should double-check their work using the code change notifications (see Practice 14, *Send Code Change Notifications*, on page 97). Monitoring the code change notifications gives you an easy, noninvasive way to keep an eye on any member of the shop without sitting in their office and looking over their shoulder.

At times you will be engrossed in a problem and would be completely derailed by stopping to participate in a code review. Trying to get your head back into the problem would cost you a great deal of time.

[9] Proverbs 27:17, NIV

Virtual Code Reviews

Over time, you will learn what types of things specific reviewers will look for. For instance, Jared once wrote an intricate piece of code and had it working to his satisfaction. He then did a "virtual review," trying to spot what two of his most senior co-workers would target. After implementing the changes he thought each of them would have suggested, he had them actually review the code. They loved it! The three of them had reviewed so much code together that he was able to analyze code from their point of view. Jared had learned what two developers (with many more years experience than him) valued and was able to use that experience to improve his own work. That's why you do code reviews; you're not only building good product but also building good developers.

(Remember the discussion on interruptions? See Practice 11, *A Tech Leads*, on page 68.) If someone comes in asking for a code review (or for anything for that matter) when you're immersed in a problem, tell them you are "deep" right now and have them come back later. On the other hand, if you're the one looking for a reviewer and someone says they're deep, either go away and wait for them to get to a good stopping point or find someone else.

Much of the work related to software development is mental—getting our heads wrapped around a problem and staying there until we get it solved. It's not an insult to ask someone to come back later when you're in a situation that requires concentration. In some shops this concept comes naturally, but in others it seems to be completely foreign. It should always be okay to ask someone to come back in thirty minutes or after lunch.

TIP 17

It's okay to say "later"

Your management must *require* code reviews. If there's no management buy-in, no one in your shop has any official motivation to participate. In other words, if no one has been told to help you, they probably won't make time to do it, especially when deadlines are tight.

If your shop doesn't have a mandatory code review policy, you can still ask your teammates for code reviews. The entire team won't get the benefit, but your own code will improve. The people who review your code will also learn the benefits of code reviews over time.

Don't wait too long for any one person to have the time to do a review; walk around the shop, and find somebody who's not deep into a problem. Walk to the other side of the building if you have to, but find someone. If it's someone you haven't used for a code review before, this is a great opportunity for them to see what you do.

These quick code reviews promote mentoring without the overhead of a formal program. By varying the developers you work with for code reviews, you get the benefit of a variety of developers' experience and expertise. Each reviewer will point out different ways to solve the same problem. Some better, some not, but all different.

TIP 18

Always review all code

The goal is to learn how to think creatively while also improving your product. Learn to look at your own problems from different points of view. These short code reviews will become second nature over time, and just like microwave ovens, you'll wonder how you ever survived without them. Practical discussions of algorithm analysis or resource constraints are lessons that are taught (and remembered) because you have a practical and immediate application.

Rather than academic book learning or certifications, you'll be sitting at the workbench of various craftsmen (some masters, some apprentices) learning a little bit from each one, adding their tricks to your own, until one day you are one of the master craftsmen yourself.

How to Get Started

Code reviews are great tools! Once you get in the habit, you'll wonder how you ever wrote decent code without them. Use these tips to get started:

- Be sure everyone understands the type of code review you have planned. Review frequently on smaller blocks of code. Don't wait for weeks, accumulating hundreds or thousands of lines of changes. No MAD reviews for your team!

- Have one of your senior team members sit in on each code review for the first few weeks or months. This is a great way to share knowledge and get the reviews on a solid foundation.
- Make sure your code reviews are lightweight. It's better to review too little code than too much. Having two overlapping reviews is better than having one larger one.
- Introduce a code change notification system (see Practice 14, *Send Code Change Notifications*, on the following page) at this time. It's a great complement to your code reviews, and it helps to remind team members who forget to ask for reviews.
- Make sure you have management buy-in before requiring all team members to participate.

You're Doing It Right If...

- Do code reviews get an automatic approval? This shouldn't happen unless everyone on the team is perfect.
- Does every code review have major rewrites? If so, it indicates a problem somewhere: either with the coder, with the reviewer, or with the tech lead (who gave the directions that the coder and reviewer are using).
- Do code reviews happen frequently? If the time between reviews is measured in weeks, you're waiting too long.
- Are you rotating reviewers?
- Are you learning from the code reviews? If not, start asking more questions during your code reviews.

Warning Signs

- Code reviews are infrequent.
- The majority of code reviews are painful.
- People avoid checking in their code because they don't want a code review.
- Team members who have reviewed code can't explain what it does or why it was written.
- Junior team members review only other junior member's code.
- Similarly, senior team members review only other senior member's code.
- A single team member is everyone's preferred reviewer.