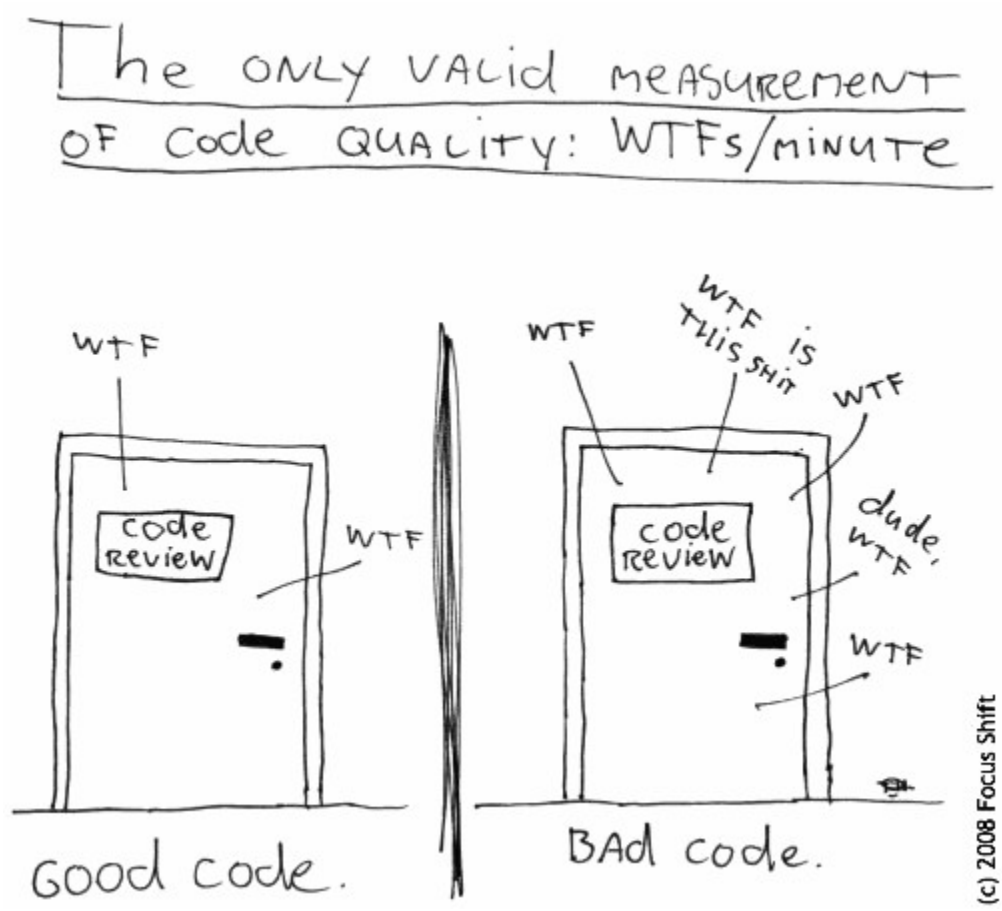
Code reviews and you



We like code reviews. While some (misguided) people may see them as introducing an extra delivery speed-bump, they do bring a number of benefits:

- To spot and fix defects early in the process and improve code quality.
- Better shared understanding of the code base as team members learn from each other
- Helps to maintain a level of consistency in design and implementation.
- Helps to identify common defects across the team, thus reducing rework.
- Builds confidence of stakeholders about technical quality of the execution.
- Uniformity in understanding helps interchangeability of team members in case of non-availability of any one of them.
- A different perspective. “Another set of eyes” adds objectivity. Similar to the reason for separating your coding and testing teams, peer reviews provide the distance needed to recognise problems.
- Pride/reward. Recognition of coding prowess is a significant reward for many programmers.
- Team cohesiveness. Working together helps draw team members closer. It also provides a brief respite from the isolation that coding often brings.
- Education of junior programmers
- Better code security

So if we take it as a given that code reviews are A Good Thing then there are a few things to bear in mind:

- When do I do a code review?
- How to do a code review
  - All code should be clean and self-documenting
  - What is self-documenting code?
- Who should do code reviews
- Some things to remember
  - Don't take offence!
  - Code reviews and pair programming
  - Some tips

When do I do a code review?

At the moment we're looking at doing code reviews as part of promoting from DEV to TEST environments. Eventually we should be looking at a Gitflow based branching model, where code reviews will be performed as part of the PR from the 'feature' branch to 'develop', but until that point, DEV to TEST (although it is good for testing to get early visibility, it can only be completed against the final version. By starting testing prior to code reviews, some of the testing may potentially be invalidated by the changes resulting from incorporating code review changes).

Code that has not been reviewed won't be promoted to Prod

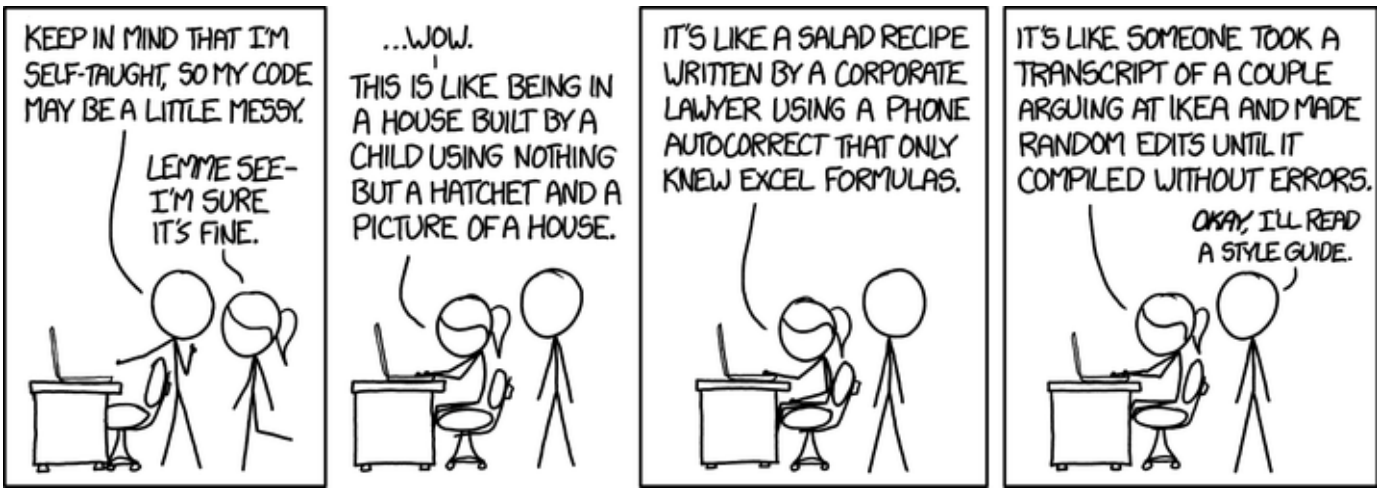
How to do a code review

Recommended process:

- Developer walks reviewer through the code
- Reviewer asks questions as they are going through it
- Developer jots down notes / suggestions / defects
- Developer amends code if necessary
- Revised code is re-reviewed
- If code is passed then it gets promoted (merged)

The main areas a reviewer is focusing on are as follows:
General unit testing - Are there unit tests? Does the code run all the tests?
Comment and coding conventions - Regardless of whether the code you're reviewing is Java, Go, PL/SQL BASIC, FORTRAN, there are conventions. For a set of conventions we work to at RI see here. There are various standards such as the SOLID principles.
Error handling - Correct use of try / catch and similar code structures
Resource leaks - Have you freed up memory you'd previously allocated, or cleared your Eden space? What impact does your code have on the Major GC?
Thread safety - Is there locking, synchronisation or atomic wrappers? Are local objects local?
Performance - does it take hours to run / is resource inefficient?
Functionality - does it do what it should?
Security - have you checked for insecure authentication, sql injection, etc?

All code should be clean and self-documenting



Code is very rarely written once and then forgotten about. Most of the time you, or more likely someone else, will at some point need to work on it again. And for that it's important that your code is clean and easily understandable. Clean code is code that is easy to understand and easy to change, i.e.

- It is easy to understand the execution flow of the entire application
- It is easy to understand how the different objects collaborate with each other
- It is easy to understand the role and responsibility of each class
- It is easy to understand what each method does
- It is easy to understand what is the purpose of each expression and variable

'Easy to change' means the code is easy to extend and refactor, and it's easy to fix bugs in the codebase. This can be achieved if the person making the changes understands the code and also feels confident that the changes introduced in the code do not break any existing functionality. For the code to be easy to change:

- Classes and methods are small and only have single responsibility
- Classes have clear and concise public APIs
- Classes and methods are predictable and work as expected
- The code is easily testable and has unit tests (or it is easy to write the tests)
- Tests are easy to understand and easy to change

If you write clean code, then you are helping your future self and your co-workers. You are reducing the cost of maintenance of the application you are writing. You are making it easier to estimate the time needed for new features. You are making it easier to fix bugs. You are making it more enjoyable to work on the code for many years to come. Essentially you are making the life easier for everyone involved in the project.

Some recommended reading material:

https://www.safaribooksonline.com/library/view/clean-code/9780136083238/
https://www.amazon.co.uk/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?ie=UTF8&qid=1526386954&sr=8-1&keywords=clean+code

What is self-documenting code?

"Code is like humour. When you have to explain it, it's bad"

Self-documenting code is code that is obvious in its functionality without having to rely on comments to explain. It will have attributes such as:

Meaningful Names - Use intention-revealing names. Choosing good names takes a little time but saves more than it takes.The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

"You should name a variable using the same care with which you name a first-born child."

Class Names - Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

Method Names - Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and prefixed with get, set.

Pick One Word per Concept - Pick one word for one abstract concept and stick with it. For instance, it's confusing to have fetch, retrieve, and get as equivalent methods of different classes. How do you remember which method name goes with which class? Likewise, it's confusing to have a controller and a manager and a driver in the same code base. What is the essential difference between a DeviceManager and a Protocol-Controller?

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

Function arguments - A function shouldn't have too arguments: Keep it as low as possible. When a function seems to need more than three or four arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not.

Now when I say to reduce a function size, you would definitely think how to reduce try-catch as it already makes your code so much bigger. My answer is make a function containing just the try-catch-finally statements, and separate the bodies of try/catch/finally block in a separate functions. If a function has 'try' keyword then it should be the very first keyword and there should be nothing after the catch/finally blocks.

Comments - if you are writing comments to prove your point, you are doing a blunder: Ideally, comments are not required at all. Our code should explain everything. Modern programming languages are English-like through which we can easily explain our point: Correct naming can prevent comments.

Who should do code reviews

Ultimately this is up to the team. Personally I don't think you should have fixed reviewers but instead you should vary it; a dev picks up a new user story from the board, and agrees a code reviewer from the team. This could be someone who has worked on a similar user story before, someone who specialises in the particular techniques that are going to be used, or maybe even a more junior developer for whom going through the code review can be part of a mentoring / training exercise.
As a rule though choose someone who has the skillset needed; there's not much point asking someone to review your C# if they've never done it before.

Some things to remember

Don't take offence!

You shouldn't have ego invested in your code - We are all talented devs here, and there will be some things that others are better at than you; one of the things that code reviews are good for is showing you if someone has a better way of doing something - it's a learning exercise. There will often be sections that get queried, and often sections that get flagged up: It is human nature that one cannot adequately proof-read one's own work. You are not your code. Remember that the entire point of a review is to find problems, and problems will be found. Don't take it personally when one is uncovered... Accept that and see code reviews as a positive thing. Besides, hard code has more defects: Having many defects doesn't necessarily mean the developer was sloppy. It might be that the code itself was more difficult, intrinsically complex or located in a high-risk module that demands the highest quality code. In fact, the more complex the code gets the more we'd expect to find defects, no matter who was writing the code. Indeed, the best way to look at this is to turn it around: If you knew a piece of code was complicated, and a reviewer said they found no flaws at all, wouldn't you suspect the reviewer of not being diligent? If I presented this essay to an editor, even after multiple passes at self-review, and the editor had no comments at all, shouldn't I suspect the editor of not having read the text?

Code reviews and pair programming

Because you've done pair programming, it means that your code review has already been done, right? After all, another pair of eyes has been looking at your code as you've been writing it... Wrong! Working as a pair means that you both develop a cognitive bias - you lose your individual objectivity. Regardless of whether you developed feature by yourself, as a pair, or in a mob, it needs to be peer reviewed by a different set of eyes - that alternate perspective is essential for an effective review.

Some tips

Don't rewrite code without consultation. There's a fine line between "fixing code" and "rewriting code." Know the difference, and pursue stylistic changes within the framework of a code review, not as a lone enforcer.

Critique code instead of people. As much as possible, make all of your comments positive and oriented to improving the code. Relate comments to local standards, program specs, increased performance, etc.

Treat people who know less than you with respect, deference, and patience. Nontechnical people who deal with developers on a regular basis almost universally hold the opinion that we are prima donnas at best and crybabies at worst. Don't reinforce this stereotype with anger and impatience.

The only true authority stems from knowledge, not from position. Knowledge engenders authority, and authority engenders respect – so if you want respect in an egoless environment, cultivate knowledge.

Ask questions rather than make statements. A statement is accusatory. "You didn't follow the standard here" is an attack—whether intentional or not. The question, "What was the reasoning behind the approached you used?" is seeking more information. Obviously, that question can't be said with a sarcastic or condescending tone; but, done correctly, it can often open the developer up to stating their thinking and then asking if there was a better way.

Avoid the "Why" questions. Although extremely difficult at times, avoiding the"Why" questions can substantially improve the mood. Just as a statement is accusatory—so is a why question. Most "Why" questions can be reworded to a question that doesn't include the word "Why" and the results can be dramatic. For example, "Why didn't you follow the standards here…" versus "What was the reasoning behind the deviation from the standards here…"