Code **Reviews**

A Step-by-Step guide to a better world by Manuel M. Huber

Code reviews are a widely spread practice in software development to ensure high code quality and are one of the cheapest ways to find and fix bugs. Like “Agile Development” there is no “one correct way” to do it but many of the biggest companies like Google have some form of code review deeply integrated in their software development process. If you don’t do code reviews already you should.

Here’s how you can go about getting started with code reviews in your projects:

1. **Win Hearts and Minds**
    a. Inform your team / manager of the benefits of code reviews
    b. Address existing doubt and criticism
    c. Ensure your team wants to do code reviews
2. **Set your Goals**
    a. List & prioritize your goals as a team
    b. Automate & delegate as many of these items as possible
3. **Define a Process**
    a. When do we review?
    b. What do we review?
    c. How do we review?
    d. Who reviews?
4. **Do it!**
    a. Teach Best Practices
    b. Avoid common pitfalls
    c. Reflect & adapt your process

Step 1: Win **Hearts** and **Minds**

You'll need the support of the developers to ensure code reviews are carried out with the necessary amount of diligence. But you'll also need the support of your manager and or customer since they will be paying for the time required to review code.

You can convince a lot of developers of code reviews by telling them "*That's what they do at Google!*". If this doesn't work, try the following:

1. You no longer have to worry about being solely responsible for breaking something – somebody else reviewed it and also thought it was ok!
2. You no longer have to worry about that one co-worker messing up your code base – somebody is going to review it first!
3. Reviewing code is like refactoring – but instead of you refactoring old legacy code you just tell the author to refactor it themselves before it becomes legacy code!
4. You can show off your fancy coding skills and learn a new trick or two yourself!

Code Reviews don't actually create more work they just make sure the work is being done **right now** while everybody still knows what and why the code does what it does instead of a year down the line when it is considered legacy code. There's no easier way to

- Improve Code Quality (thus reducing the need to refactor later)
- Find bugs (and fix them right away)
- Share knowledge of the code base
- Help new team members learn and contribute

Your manager or customer is probably going to want to see some numbers. The benefits of the developers still apply to them as well (since developer time is money) but they probably need some graphs and numbers to justify the investment. Show them the page "The Price of Quality" for some basic information about code reviews and one of the many real life study that show the savings due to code reviews.

Answer these common objections if they arise:

- "It slows us down too much"
    - While review adds time to each feature the reduced error rate cause release cycles to shorten
- "It's too expensive"
    - The reduced number of bugs pays off immediately and the reduced technical debt will pay off in the long run. It's an investment that will show significant return in long term projects.

Step 2: Set **Goals**

The goal of this step is to

1. Make code quality present in everybodies mind
2. Have benchmarks of quality explicitly written down
3. Set up (automated) ways to reduce menial tasks during reviews

Gather your dev team and list all quality aspects that are important to you (use the list below as inspiration). Then see which of these you can accomplish without reviews. Automate as much as possible. The rest of these items will be your checklist for the actual code reviews.

Many of these items require something else first. To automate unit tests you'll probably need a continuous build server. To ensure a consistent architecture you'll need an explicit architectural design and so on.

You should probably have:

- ☐ build server with code-quality tools
- ☐ style guide (make use of existing guides!)
- ☐ architecture design
- ☐ Dos and Don'ts list specific to your project / code

Here is a non-exhaustive list of things to consider:

- bugs
- code style (formatting, naming, …)
- readability / maintainability
- architecture (single responsibility, dependency inversion, interface design, package structure, …)
- code conventions (Do you have Util classes? Where and how are constants defined? …)
- best practices (specifically for your project / languages / technologies / …)
- functional correctness (all requirements fulfilled? Edge Cases checked?)
- UI-Design
- test coverage (all critical aspects tested? Code quality of the tests good?)
- security / legal requirements
- performance
- documentation

Step 3: Define your **Process**

Now you should write down how you're going to review. One of the most common methods are regular tool-based reviews but there's no "one correct way". Feel free to mix and adjust these methods as you need.

At the end of this step you should have a process diagram that explicitly shows every step of the review process and can be understood by new employees without the need for any explanation. At the end of this document there is an example process for reference.

When & what to review?

The decision about when to review which piece of code is one of the most fundamental of the entire process. Since Code Reviews are such a great investment you should try to get your team to commit to as much review time as possible – preferably a regular review integrated with your existing process.

On-Demand Review

You review a specific piece of software when there's a significant reason: A team member feels the quality of this code is particularly low, it's critical to the core use case of the software or the author asked for a review.

+ Low amount of time investment
+ Improves quality of critical areas

- Only a small portion of code is reviewed
- Review experience is gained very slowly, increasing the time per review needed

Timeboxed / Random Review

Every developer has a set amount of time per week/sprint reserved for reviews and picks the code to review randomly or at will.

+ Costs are very predictable
+ Even the chance of being reviewed causes developers to write higher quality code

- Some code may never be reviewed
- Picking code can cause friction between developers

Regular Review

The review is an integral part of your development process. Most commonly this is done with git-branches where new code is only allowed to be merged after a successful review.

+ Highest error detection rate
+ Review efficiency improves rapidly through experience

- Highest amount of time invested

How to review?

Decide how exactly the review process will be executed. Take special consideration about how this will integrate with your existing process and tools.

Group Review

The team members prepare the code beforehand and discuss it in a moderated meeting. Often members will have a different role and present their findings which are then recorded.

+ Very thorough
+ Many different perspectives
- Extremely expensive
- Disrupts the workflow of the entire team

Over-The-Shoulder Review

The author explains the code to the reviewer who is either physically looking over their shoulder or via a shared screen. The reviewer gives notes and issues are fixed immediately or directly after the review. After the review the author decides when the code is ready.

+ Fast and simple process
+ Discussions can be solved instantly
+ Good way to teach and learn
- Author guides the review which reduces objectivity of review
- Both author & reviewer need to be present simultaneously
- Hard to track and enforce

Round-Robin / Tool-assisted Review

The author checks the code into a review system (often included in the VCS) and a reviewer reviews it when convenient and the author fixes or comments on these discussions. Multiple review-change-iterations can take place. The review is done when the reviewer accepts the proposed code.

+ Thorough and traceable process
+ Many tools offer great functionality and integration with existing infrastructure
+ Convenient & undisruptive to workflow
- Multiple review rounds can drag on

Pair Programming

While not technically a review type this can be especially effective when teaching unexperienced developers since instead of having to work through a large amount of comments you can ask & answer questions instantly.

+ Excellent way to transfer knowledge
+ Can prevent fundamental problems while the code is being written
+ Builds team spirit
- Time consuming
- Reviewer is partial author which reduces objectivity

Formulate Process

Now you need to make a flowchart of the entire development-review process. Some of the important things to answer is:

- What tools are we going to use?
- How do reviewers know there's something to review?
- Who decides which reviewer reviews what?
- Who, how and when fixes issues from reviews?
- Are fixes reviewed again?
- Who can close a review?

This is important to ensure everybody is on the same page and it's clear who is responsible for what. Don't worry about getting it perfect the first time around. Just start with a process that seems about right and adjust it as you notice flaws.

Take special consideration about how the process integrates with your existing workflow. Code reviews should be as easy as possible. Try to avoid manual work (like sending code via E-Mail) or too much context switching (for example because the review tool is not integrated with your VCS system).

About **tools**:

The main activity will be reading code and writing comments to specific lines of codes which is supported practically every review tool. While navigation comfort or IDE support are especially useful for larger reviews the most important aspect is ease-of-use. People will be very reluctant to do code reviews when the review is already tedious to start / close – so integration is (usually) more important than extra features.

If you use GitHub, GitLab or BitBucket use their integrated review tools to get started. Otherwise use what people in your organization / team recommend or simply pick one of the many tools based on their integration, pricing and features: UpSource, Collaborator, CodeBrag, Microsoft Team Foundation Server, ...

Step 4: **Do** it!

Here are some best practices:

1. Automate as much as possible
2. Treat reviews as high priority
3. Be generous with code examples (especially when trying to teach through review)
4. Try to keep the reviews small – preferably under 1.000 loc

Pitfalls to avoid:

1. Making it personal

   Keep the language neutral. Avoid the use of “you” since it often sounds accusatory. Make simple, neutral comments like “Typo: Fobar -> Foobar” whenever possible. Don’t command the author, make suggestions. Be aware to not pass off your opinion as fact. And try to resolve any matter of opinion (like code style) with something you all agreed on beforehand (like a style guide).

2. Comment flooding

   When the same issue persists throughout the entire code (like wrong naming convention) don’t comment on every occurrence but note it in a single comment. In general: Your goal is not to create perfect code, but to make the code better. So don’t nitpick too much – especially with new developers. They will get better over time but too many comments on one review will discourage them.

3. Overstepping the scope of the review

   Often you will see room for improvement on code that’s not been part of the submitted review. Don’t make it the authors responsibility to fix these things “while they’re at it”. You’re reviewing the changes they made and nothing more.

Lastly: Do a **retrospective**!

Your process will most likely not be perfect and that’s normal. Maybe your tools are too inconvenient and you need to look at that. Or the style guide isn’t complete. Maybe you’ve noticed that reviews for trivial changes are blocking you unnecessarily so you need to adjust your criteria when to review. Whatever it is – talk about it and fix it.

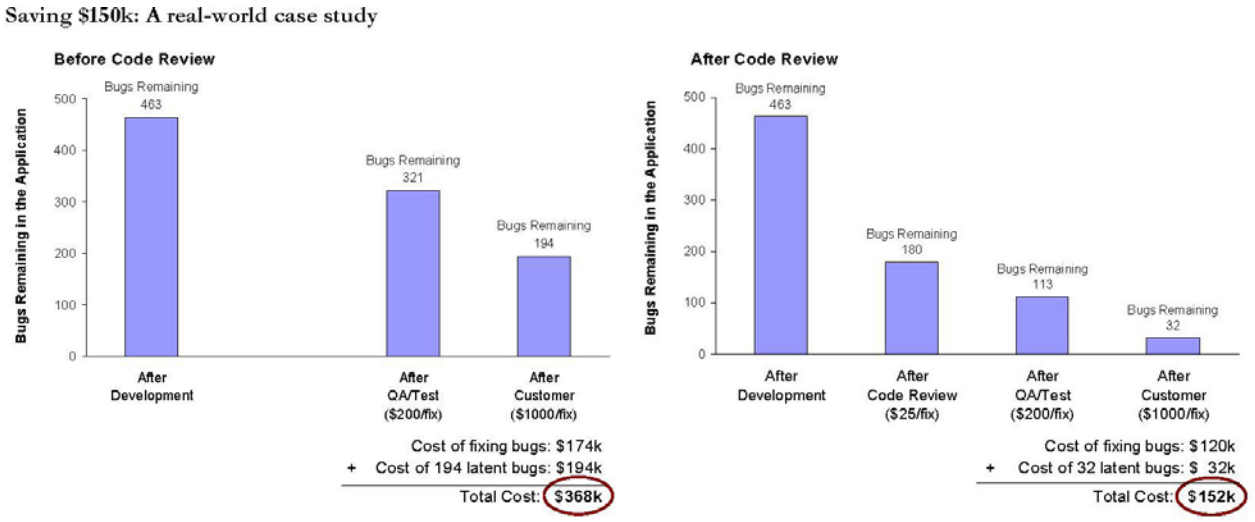
The **Price** of quality

Code quality is never free. You can either pay it **now** or pay with **interest**.

While the investment in code quality is an increase in costs it will pay off in the long run. So-called "*Technical Debt*" (caused by low quality code) increases the cost of every bugfix and new feature since a low-quality code base makes changes and additions more difficult and time consuming. One of the cheapest ways to pay of this debt is to do so right away: During a **Code Review** a second developer looks over the code to spot any mistakes before the software goes into production or even QA.

Several case studies with significant project sizes have shown that regular code reviews are one of the most cost-effective ways to improve code quality and find errors.

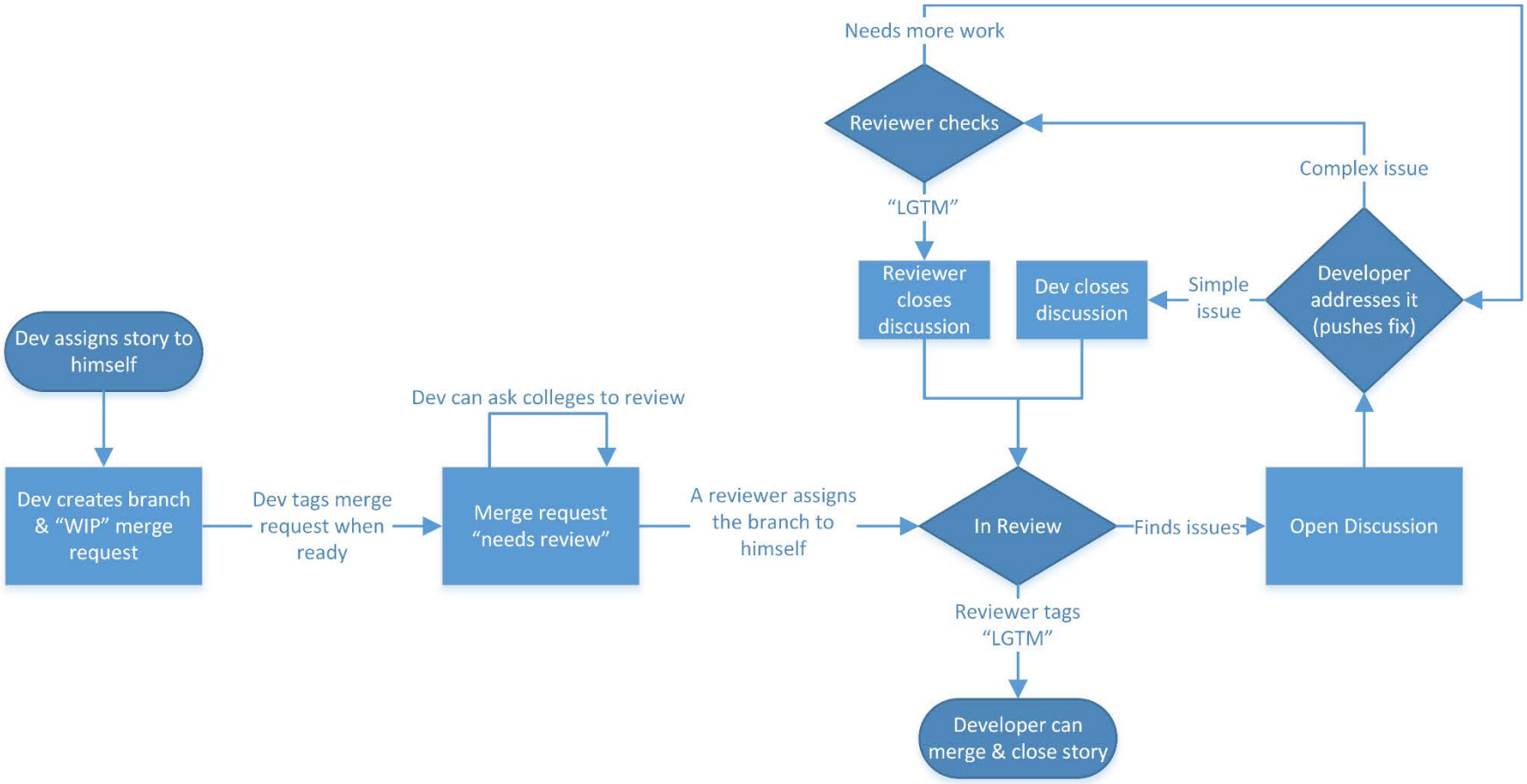
One study calculated the savings on a project of 10 developers and about 10.000 lines of code. They sent the project to Quality Assurance (and subsequently customers) once without code reviews and once after being reviewed by another group of developers first. The result was that the reviews not only **halved the cost** of bug fixes but also delivered a vastly **more bug-free** product to customers:



Many big companies, like AT&T, IBM and Google introduced some form of code review and have found that error rates drop by up to **90%**.

Besides the financial benefits code reviews also:

- Spread knowledge throughout the team making personal changes less troublesome
- Make on-boarding new developers easier
- Shorten release-cycles due to less bugs and shorter QA phases


Dev assigns story to himself
Dev creates branch & "WIP" merge request
Dev tags merge request when ready
Merge request "needs review"
Dev can ask colleges to review
A reviewer assigns the branch to himself
In Review
Finds issues
Open Discussion
Reviewer tags "LGTM"
Developer can merge & close story
Needs more work
Reviewer checks
"LGTM"
Reviewer closes discussion
Dev closes discussion
Simple issue
Developer addresses it (pushes fix)
Complex issue