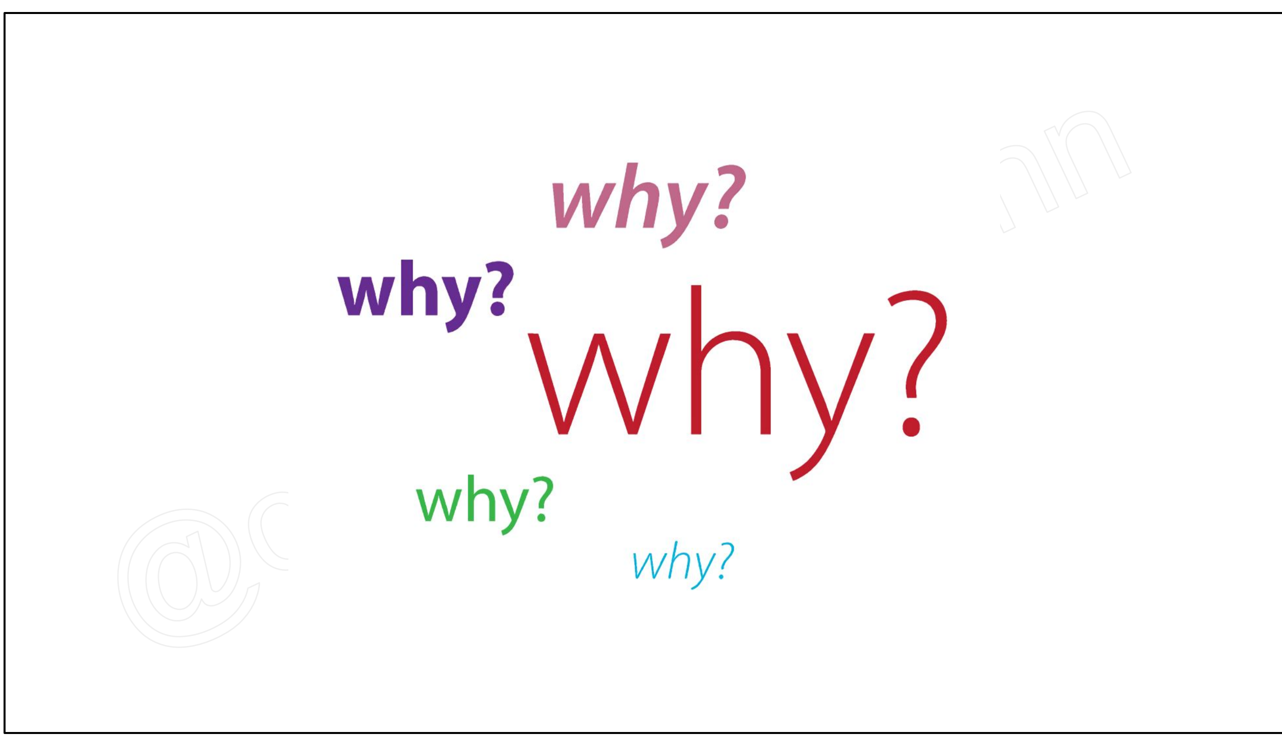
What's up! My name is Curtis, and welcome to my video course — Master the Code Review. Forge a better process. Give better reviews. Write better code.

I'll introduce myself later. Let's jump right in. We'll start with…



WHY did I build this?

Code review is such an important piece of a software engineer's day-to-day. On a daily basis, part of our day is dedicated to preparing code for review, or reviewing code. Yet there is very little material out there about code reviews. There are style various style guides specific to language and framework. There are some blog posts here and there. But there is very little content that gives actionable advice on how developers can about succeed in a code review environment. And there is no denying its importance.

A few years ago, Google published some code guidelines on how to perform a code review. The guidelines emphasize the importance of various non-technical skills, as shown here, like be courteous. There was a large response in Hacker News — let's take a look at this comment. "In CS engineering classes, where presumably we would learn to become great engineers, I don't recall learning about any of this, and instead I remember the emphasis being on technical knowledge and accomplishment. I'd probably have been a better engineer in my early career if I'd understood how much the ability to exchange clear, constructive, and nonthreatening critique with peers actually mattered."

This comment in particular resonated with me. I think technical skills are important. But I think many educational programs underemphasize on the social aspects of software engineering. This isn't necessarily an indictment on computer science curriculums. I think there is a saturation of courses and tutorials teaching others about

the latest and greatest tech stack. There aren't enough courses covering framework agnostic principles, soft skills and technical skills required to succeed as a software engineer.

Yet, these skills are directly intertwined with a developer's success at a company. Recently, Dropbox publicly published their guidelines and performance expectations for software engineers at various levels. Many big tech companies have similar guidelines. Let's take a look at what they're saying:

From the IC2 guidelines:
"I have self-awareness about my strengths and areas for development"
"I translate ideas into clear code, written to be read as well as executed"

Dropbox leveling guidelines for IC3.
"I solicit and offer honest and constructive feedback that is delivered with empathy to help others learn and grow"
"I ensure high code quality in code reviews."

IC4:
"I identify and support areas of growth for my teammates that take into account their skills, backgrounds and working styles"
"I model integrity and a high standard of excellence for my work. I leverage this to set and hold the bar for quality and best practices for my team (e.g. via code and design reviews)"

Notice how none of these things said "master React" or "excel at the JavaScript" — while it's great to learn the ins and outs of these technologies if your team is using them, they hard skills alone aren't enough to level up in the industry.

This course is different…

Why should I watch this course?

📈 LEVEL UP (your skills)

🎒 Technical skills AND soft skills

1️⃣ **Forge a code review process** to help your team ship better software, faster.

2️⃣ **Give better reviews** that drive code quality and elevate the skills of your peers.

3️⃣ **Write better code** that gets approved in the first review.

@curtiseinsmann

This course is designed to help you level up your skills. I can't guarantee promotions. Each company follows a different process — not all of them have the same criterion as the Dropbox ones I showed you. But what I can say is that this course will make you a better engineer. It'll put you in position to grow.

It will teach you to do this through soft skills, technical skills. These are language agnostic principles that will be applicable regardless of language and framework you're using. If you commit code to a version control system, this course is for you. It'll teach them through the paradigm of something we do every day, which is code review.

So I will teach you to master all 3 dimensions of code review. What are those? At a high level, there are three aspects — the process your team is following, the reviews your giving, and the code you're preparing to go through review.

Module 1 of this course:
Forge a better process…
You will learn:
- signs of a bad code review process, which many teams have but just aren't seeing
- what a good code review process looks like, so you can empower your team to ship faster
- how to establish a code review process for your team
- example of an effective code review process that you can use as a model

Module 2 of this course:
Give better reviews…
- what to look for in a code review, so you're prepared to spot things
- how to perform a code review, step by step
- write effective code review comments to level up your peers and build relationships
- an example code review

Write better code…
- principles to write better code
- step-by-step process to write and produce a better code review
- address code review feedback
- example authored code reviews

This course will be useful to developers of all levels, but the most useful will depend on your level. Developers who just broke into the industry will be most interested in module 3, write better code. Experienced junior and early mid-level developers will be interested in module 2, give better reviews. High mid-level and senior level developers will be interested in module 1, forge a better process.

You'll notice that the order of these topics is opposite of what you'd expect, if you think about a level hierarchy. I originally intended to give the modules in the reverse order. However I noticed that it doesn't matter how good of a programmer you are — if the process is bad, your success will be largely out of your control. You need to know if you're participating in a dysfunctional process, and cover gaps accordingly. Regardless of what level you are, you'll get value out of all modules.

Why should I listen to you?

550+ code reviews authored

90% (last year) approved in 1st review

850+ code reviews reviewed

📈 helped dozens of developers level up

[personal opinions only!]

Now finally, why should you listen to me? Who am I?

The short story is I've done everything I said I was going to teach you in the last slide — and I've done it many, many times. I've mentored and taught others how to do it many, many times. It works.

I am a freelance software engineer. Right now my major client is Gumroad (likely the same platform from which you bought this course). Gumroad is known for its completely asynchronous, no meeting environment. So in order to work there you have to be stellar at asynchronous collaboration, writing and of course — code reviews.

Before Gumroad I was at Amazon Web Services for 6 years. You can see me there with my Amazon shirt collection, which I was quite proud of. I wore an Amazon shirt to work every day, even during the pandemic.

When I started out I would regularly receive 50+ comments on my code reviews. My code was bad, and it was pretty embarrassing. I felt like I didn't belong. I often went through 8+ revisions in order to ship code. Over and over again, I had to submit my code for review, and over and over again I would get comments.

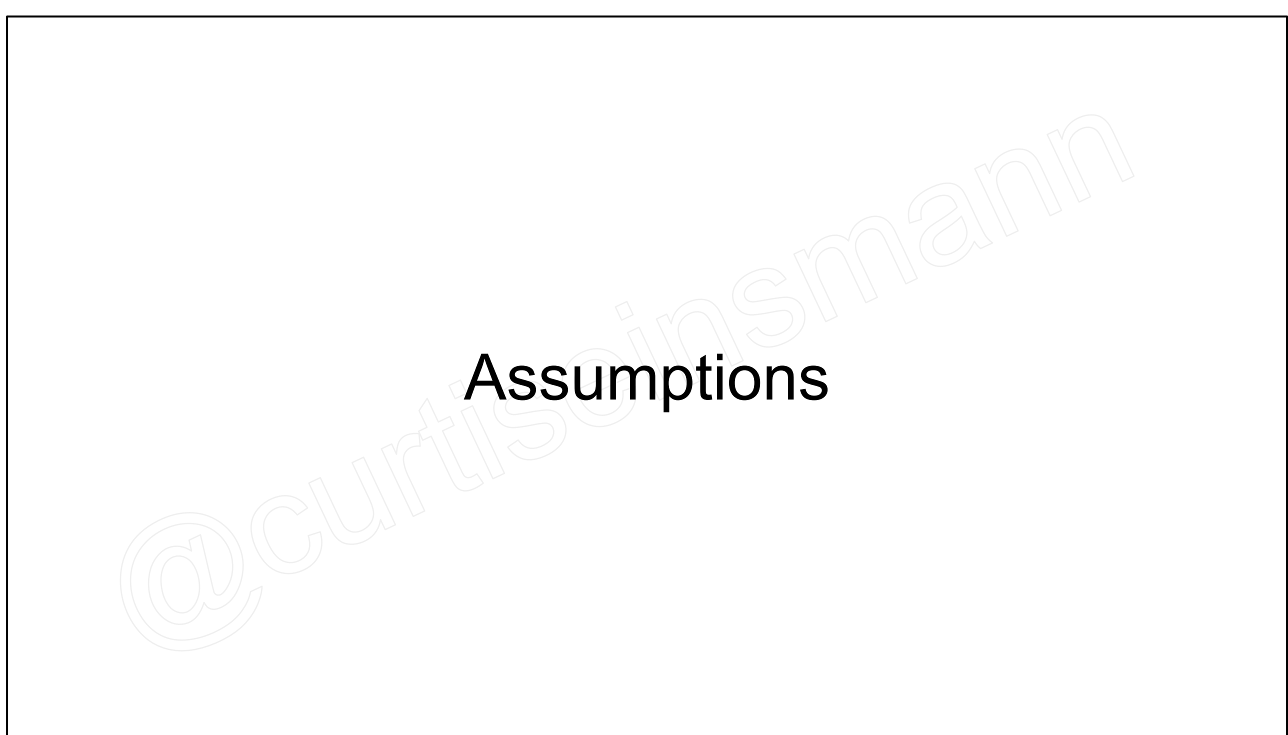
Since then I've leveled up. I spent about 6 years at AWS. I went from intern to lead software engineer. I designed developed deployed and maintained large scale software systems.

Numbers tweet:
https://twitter.com/curtiseinsmann/status/1410282734493204481?s=20

I’ve authored 546 PRs in production at Amazon.
It is a company of extremely strong opinions. Developers have high opinions and expectations on code quality.
I got better and in my last year, 90% were shipped on the first review.

I’ve reviewed over 845 PRs at Amazon.
I’ve helped people get promoted. I’ve been an instructor for internal bootcamps at Amazon.
I’ve mentored teams and organizations to refine their code review processes.

Keep in mind that this is opinion based. I’m not representing my employers, past present or future. I only talk about things I have experienced. This course is 100% bootstrapped, and my own creation. So you won’t find this material anywhere else.



Now let's talk about some assumptions before we get started. This is general housekeeping before I get into the content.

You’re familiar with “some code that gets reviewed”

- **Pull Request (PR) — GitHub ⭐**
- Merge Request — GitLab
- CL or changelist — Google
- **CR or Code Review** — Amazon
- Diff — Facebook

@curtiseinsmann

As developers most of us are familiar with “some code that gets reviewed.” You write some code, and it’s viewable as a change that a teammate will review. This can be one commit, or multiple commits. Usually it spans across many files of source code.

It wasn’t until I started making content about code reviews that I realized that different organizations call this different things. It makes my life harder as a content creator, to be honest.

But you’re likely familiar with one of these things. <enumerate>

Most of the time I’ll be referring to this “thing” as a Pull Request or PR. Sometimes I will refer to it as a Code Review. This is because many people are familiar with GitHub. But the principles I talk about in this course will apply regardless of whether you use PRs, MRs, CLs, etc.

Probably because you’re maintaining it.

Some people really didn’t care about the code they wrote because they’re just building out a prototype and throwing it over the wall to somebody. Or maybe they’re at a startup that has a “move fast and break things” mindset — which I don’t agree with, but that’s besides the point.

Some companies have different ways of creating incentives for their coders to desire code quality. My assumption here is that you’re maintaining your code, and have some sort of incentive for its quality. Maybe you want to develop features faster, maybe you want to debug it faster, or you want it to be easier to maintain.

Your software doesn't have zero margin of error.

@curtiseinsmann

I'm mostly going to assume that you're working on something that's intended to make money, or software that helps other people make money. But human lives aren't in danger if it doesn't work. E.g., if you're writing software that'll be used on a spaceship or a self-driving car, you're likely going to need more scrutiny in your code review process than what I suggest in this course.

A note on open source — this course is not intended to targeted towards open source developers. It's meant for developers actively writing software for a company with which they have employment. However there are some principles that could and do apply to open source development, since code reviews are a very common process there.

Is this course for open source developers?

@curtiseinsmann

A note on open source — this course is not intended to targeted towards open source developers. It's meant for developers actively writing software for a company with which they have employment. However there are some principles that could and do apply to open source development, since code reviews are a very common process there.

You’re submitting your code to a Version Control System (e.g., GitHub)

@curtiseinsmann

You’re submitting code to a Version Control System.

I’m going to assume you know git fundamentals such as pulling, branching, merging.

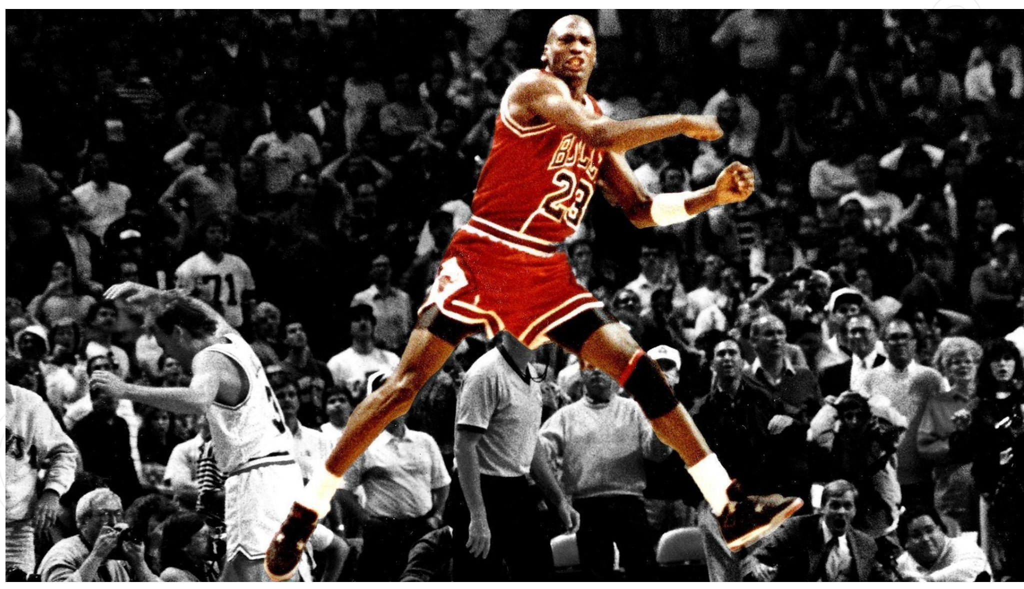
I’ve confirmed that GitHub is a type of VCS:
https://hackernoon.com/top-10-version-control-systems-4d314cf7adea

This is a video course.

@curtiseinsmann

Videos are numbered. No interaction, watch the videos in order. You can stream it from any device that you can get internet. If you have questions, just simply reply to the confirmation email that you received when you bought this course.

End of Part Zero! 🐂



Congrats! You’ve reached the end of Part Zero! Here’s a picture of Michael Jordan.

If you learn nothing from this course, learn this: MJ is the best basketball player of all time. Not LeBron, not Kobe, not Kareem, not Wilt, or any other player. At the end of each high level module, I will give you reasons why Michael Jordan was the best basketball player ever. And I will relate them to how you can succeed in code reviews.

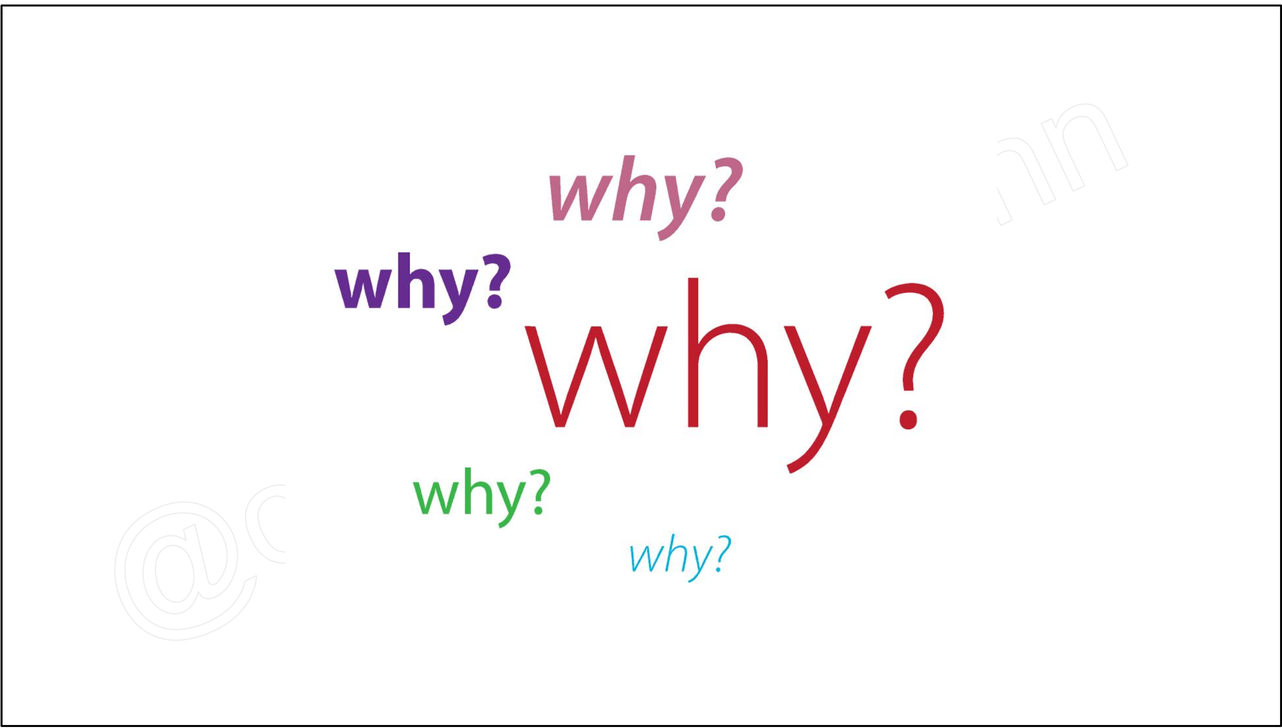
With that being said, let’s proceed to the next module. We’ll learn to forge a better code review process. Go ahead and open up the next module, and I’ll see you there!

Part 1: Forge a better code review process

Empower your team to ship better software, faster.

@curtiseinsmann

What's up! Welcome Part 1 of Master the Code Review — Forge a better code review process. In this module you'll learn how to empower your team to ship better software, faster.



Why do we care about forging a better code review process?

Earlier I showed you the Dropbox leveling guidelines. Let's take a look at the IC4 guidelines. This is your senior, and staff at some companies.

First, this one:
"I model integrity and a high standard of excellence for my work. I leverage this to set and hold the bar for quality and best practices for my team (e.g. via code and design reviews)"

As a leader of your team, you want your team to be effective. You can't write and review all the code. If you could, there would be no need for teammates. You need to forge a good process so that your team can operate effectively. You want them to ship fast and hit deadlines. You want them to maintain high code quality without your input. You don't want defects. You don't want defects.

"I define my team's priorities and secure buy-in in partnership with my manager"

The manager isn't always the one in charge, because they have other responsibilities. You're the one close to the code. You're the one that knows what technical items need to be prioritized. It's your job to get the team aligned and working towards the common goal. Code reviews are an excellent opportunity to do that.

[Back to slide] Remember, as a developer, you may be participating in a code review process. You may think your is good enough. But the things I'm going to show you here may make you think otherwise.

Why should I watch this part? (Agenda)

- Signs of a bad code review process 🚩
- A good code review process 🚢
- Example: Gumroad's code review process 🍬

@curtiseinsmann

We're going to talk about signs of a bad code review process. We'll go through some situations you may be familiar with, and you'll be able to spot gaps.

We're going to talk about what a good code review process looks like. From set up, to author and reviewer expectations.

I'm going to walk you through an example of a good code review process. Specifically, the code review process followed at Gumroad, known for its collaborative async environment.

Why should my team perform code reviews?

- 😞 Mistakes
- 👥 Distributed Ownership
- 👍 Collaboration
- 📋 Coaching
- 👩‍⚖️ Accountability

You might even be asking — why should my team perform code reviews in the first place?

DONE WELL — these things are true.

Great article: https://smartbear.com/learn/code-review/what-is-code-review/

Humans make mistakes.
Less bugs.
Can't rely on QA or tests. Good to have them, but there are edge cases bugs that we'll talk about which we can't really test in QA.
Save time — reduce the amount of work performed by QA. Reduce cognitive load of developers that have to build features on top of it.

Distributed ownership and blameless. As a developer, I actually want people to review my code. If I deploy a bug, it's very easy to say oh that person caused it. Every push is all about the team.

Better communication and camaraderie across the developers on the team. Encourage thoughtful disagreements and sharing of opinions. Knowledge gaps for developers across the team.

Educate junior developers.
Developers get exposed to problem solving strategies.
Senior developers demonstrate how to write good code and make changes.

Accountability. Code quality is important. If I'm shipping code by myself and nobody is really looking at it, my quality isn't going to be as high. It'll deteriorate. By knowing somebody is going to be looking at it, I'm going to do better work.

Save time and money. Less bugs. They are very costly to dig into as codebase grows in complexity. Higher quality code leads to easier shipping of features.
Asynchronous work helps people be independent on their own projects.

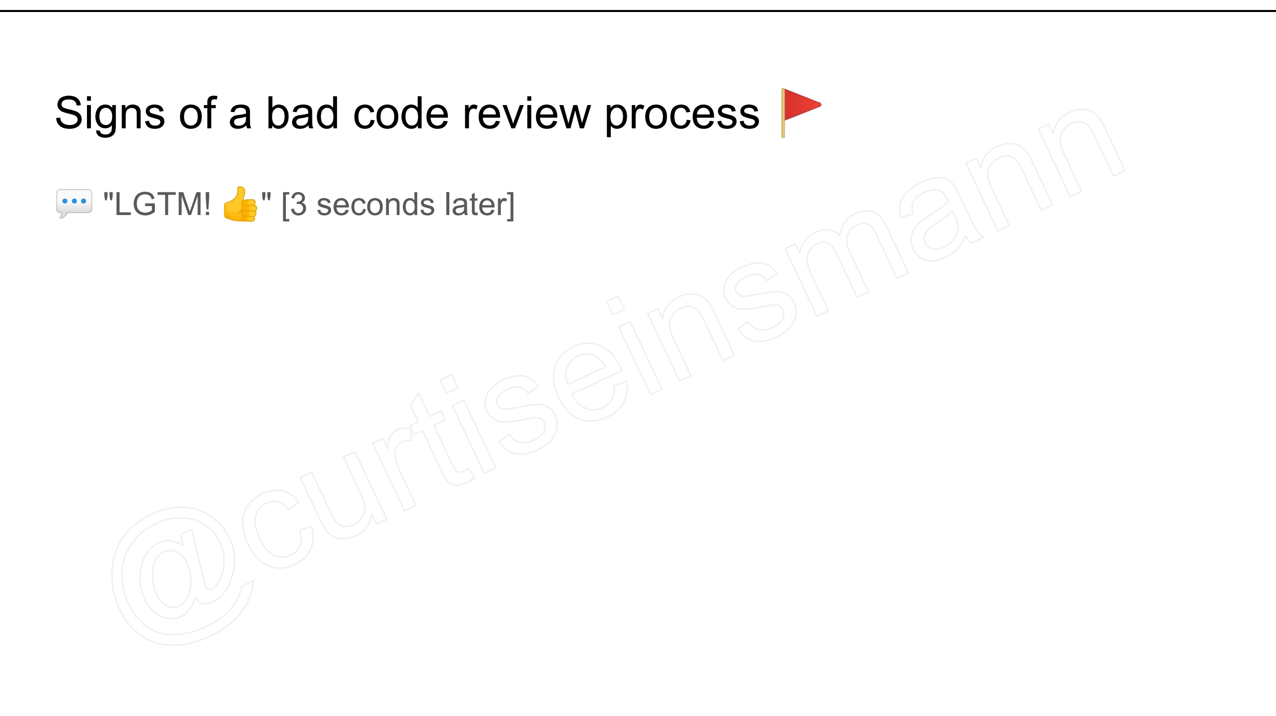
In Part 3 we're going to go over how to forge a better CR process for your team and organization.

Signs of a bad code review process 🚩

@curtisoinsmann

Welcome to this submodule — signs of a bad code review process. Maybe you're participating in a code review process, but you're not sure if it's good or bad.

We're going to start off with a bit of fun. I'm going to present some situations, and we'll see what kind of code review comments come up. I've personally seen every single one of these happen — and I'm willing to guess you've seen something similar too, if your organization performs code reviews.



You're modifying a legacy function that could break the entire app. After reading it for 2 hours, you finally understand how it works. You slowly, carefully type out 5 lines of code and open a review.

<Read comment>

This is a sign that your team is not taking code review seriously, or they're being lazy about reviewing code. Code reviews take time, and they aren't taking that time to perform a thorough review. This is a dangerous game — defects could slip through in the future.

Signs of a bad code review process

💬 "This code is horrible. There's no way we'll be able to maintain what you've written."

@curtiseinsmann

Thanks to your hard work, customers will now be able to edit widgets! You open a code review, excited and proud of what you've accomplished.

<Read comment>

This is a sign that your team is not prioritizing kindness. They talk down on each other, and may even be competing with each other. This is bad for team camaraderie, and their effectiveness suffers as a result.

Signs of a bad code review process

💬 "The existing function is unreadable. We should refactor it as part of this change."

@curtiseinsmann

Your app is broken. Customers are angry, and your ticket queue is growing to infinity. You open a 2-line code review to fix it.

<Read comment>

This is a sign that your team doesn't understand when to consider the context of time sensitivity of the change. They may be perfectionists, or on a power trip to impose their will.

Signs of a bad code review process 🚩

💬 "Here on line 34, we should..."

💬 "And on line 45, we should..."

@curtiseinsmann

A review just came in. Only two comments, lines 12 and 23. What a relief — you're so close to shipping! You quickly address the comments, and open what's sure to be the last review.

<Read comment>

This is another sign that the reviewer is not being thorough on each review. They're partially reviewing the change, not the whole thing. This leads to unnecessary iteration and churn, causing your team to ship later than necessary.

Signs of a bad code review process

- You don't have one
- You have a large queue of minor bug reports
- There are rarely any comments
- They make you anxious
- It's common for 1 PR to go through many review cycles (7+)
- Long discussion threads; back-and-forth
- Code reviews are blocked for minutiae (like style)

@curtiseinsmann

Now that we've gone through those example scenarios, let's talk about the more general signs of a bad code review process.

You don't have one. Obviously, that's bad. Humans make mistakes, even the best programmers. Bugs and flaws are very expensive, and not all of them can be caught by tests and QA. We talked about the value of code review earlier, I need not repeat it here.

You have a large queue of minor bug reports. Every team has bugs, but if two or three are cropping up every time you release a feature, it's a sign that these aren't getting caught.

They make you anxious. This is a subtle one. If you open a code review, and you're worried about what a reviewer is going to say, that's a red flag. It can mean a few things: your team is mean, they're bad at giving feedback, they're stepping on each other's toes. This leads to a dysfunctional team environment where people are unwilling to give each other feedback and listen to each other. The quality of your software will suffer.

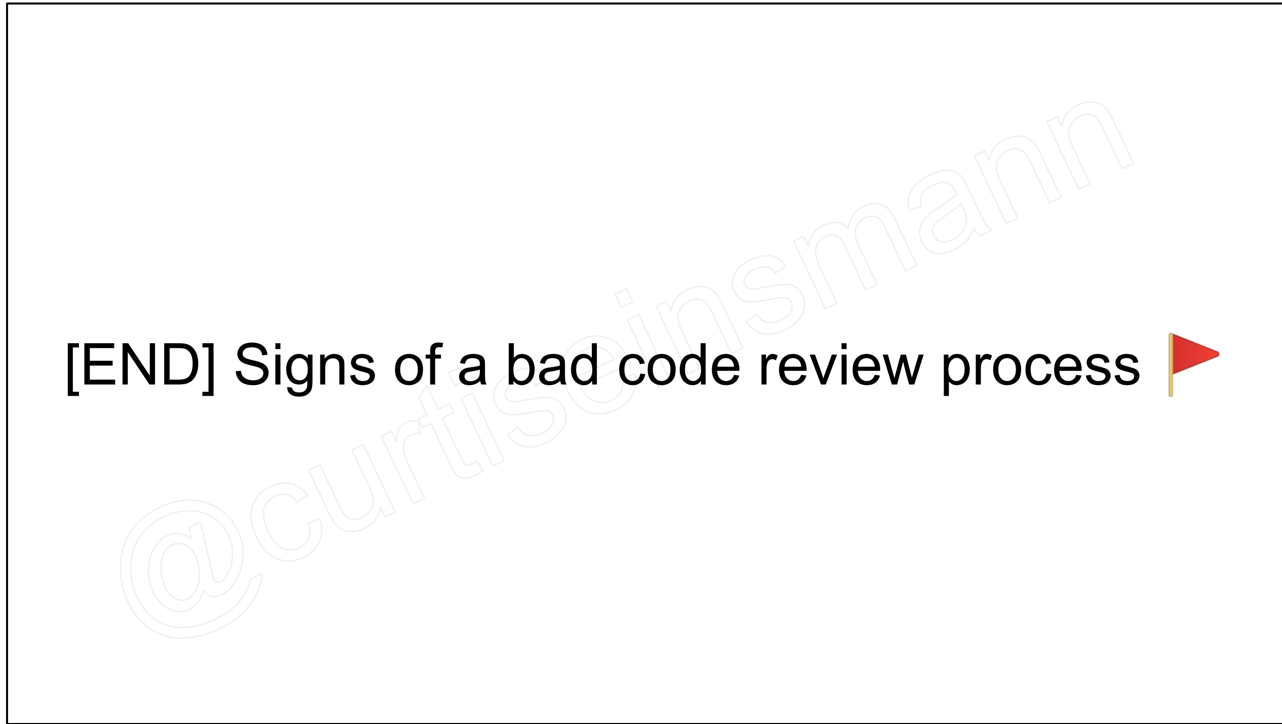
There are rarely any code review comments. If your team is approving everything right away, they may not be reviewing thoroughly. They may be lazy.

It's common for 1 Pull Request to go thorugh many review cycles. This is a sign that code changes may be scoped too large, or reviewers aren't being thorough — the

increased churn will cause your team to ship slower.

Long discussion threads; back-and-forth. Reviewers aren't communicating their concerns well enough. Leads to churn and delayed shipping.

PRs are blocked for style. Code review comments should rarely be about style — almost not at all. Style decisions should be left to the automated tooling. If there are long threads about style, your team may be focusing on minutiae, not what's important. Stepping on each other's toes.



OK! Those are the signs of a bad code review process. In the next section we'll talk about what a good code review process looks like.

A good code review process 🚢

@curtisoinsmann

Welcome to this submodule. In the last submodule we talked about the signs of a bad code review process. Now we'll talk about what a good code review process looks like.

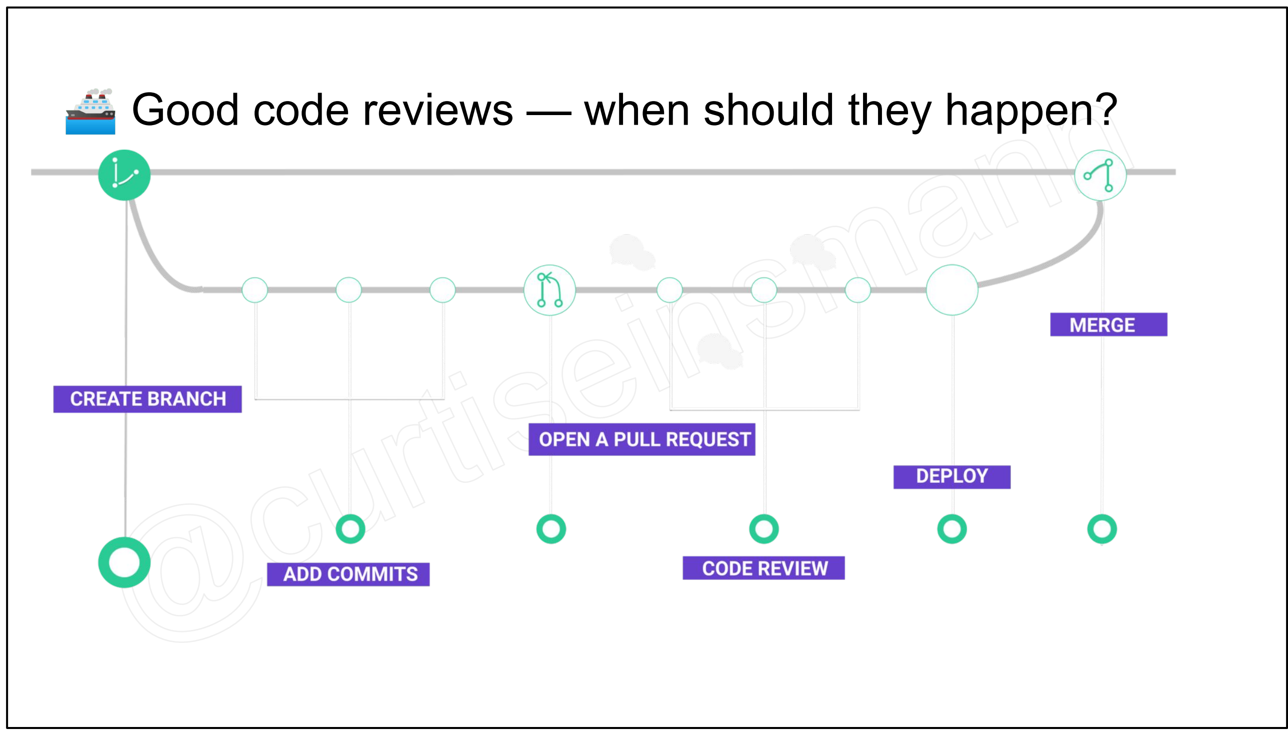


Image source: https://gitential.com/pull-requests-and-code-reviews/

So this is a diagram of what a good code review process looks like. And I'm going to walk you through this diagram.

Code reviews should happen as early as possible. Ideally, they should happen before QA. They might even need to happen after QA.

So ideally the developer should be able to do their development locally. They should be able to set up their own environment where they can test the application, including unit tests, integration tests, E-2-E tests. They should be able to integrate with dependency systems in a way that mimics the pre-production environments. Sometimes this is feasible, sometimes there are challenges.

Here I'm going to assume that you're using some sort of version control system, like GitHub. They would create a branch for a new feature they're developing. They'd make some code modifications, and commit as they please. Once they are implementing the feature or bug, and they've tested it on their own, then that's a good time to open up a Pull Request. Then they send it to their teammate for review. The code review is when the teammate looks at the code, spot checks it for flaws, readability gaps, etc.

Then after this step, it really depends on the application. This particular image says deploy. Maybe you could deploy it to another environment so that QA can look at the

change and make sure it’s up to standards and doesn’t have defects. You may even want to perform another code review after this step.

Then comes the merge. This is when the code change is deemed good to go, and is merged in with the rest of the code that runs your application. Ideally the code shouldn’t be merged unless it’s production-ready. But there may be steps before reaching production, depending on how your release cycles are set up.

So to recap, the author of the code creates a branch, makes the code changes, opens a pull request, and that’s when the code review happens. When the code is free of defects and production ready, that’s when the code is merged to the main branch.

Good code reviews — synchronous or asynchronous?

- Synchronous:
  - ✅ Good for:
    - Fragile changes
    - Large, complex changes (should be avoided)
  - ❌ Bad for:
    - Finding flaws and defects
    - Individual productivity

@curtiseinsmann

So when we get to the code review portion, should they be synchronous or asynchronous? Synchronous meaning that two developers would be looking at the code at once — the author and reviewer. Asynchronous means that author sends to reviewer, reviewer comments, author responds and addresses, etc.

Let's talk about synchronous first. When would you perform synchronous code reviews?

Good for fragile changes. Let's say we're modifying a critical part of the system that's untested, difficult to test, or would take a long time to refactor. Then it's best to put some heads together and ensure the change is safely made. Pair programming can also work great in this situation.

Large complex changes. This is if the author needs to walk the reviewer through their code. These should be avoided as we'll talk about later. But sometimes the large changes are unavoidable.

Bad for finding flaws and defects. This takes a lot of focus and time. An individual reviewer needs to have the focus and flow to dive into the code.

Bad for individual productivity. With two people looking at the code at the same time, there is less time to work on another task.

Good code reviews — synchronous or asynchronous?

- Asynchronous ⭐
  - Individual productivity
  - Greater focus
  - Writing comments forces clarity of thought
  - Fresh perspective (like a book editor) — developer sees the code for the first time

@curtiseinsmann

This is why asynchronous is recommended in most situations, and it's the one I've starred here.

Better for individual productivity. If I author a code review and send it to a reviewer, I am free to work on something else while they're reviewing, or while I'm waiting for them to review. Similarly a reviewer can work on something else while they're waiting on the author to address review comments.

Greater focus. It's much easier to focus on writing code, or reviewing code, when you're not actively talking with someone. Cuts down on multitasking.

Writing comments forces clarity of thought. If the code review is happening synchronously, people are not communicating to each other in writing. The process of writing down thoughts and feedback gives clarity to a review. Reviews are more accurate.

Async code reviews offer a fresh perspective. Like the editor of a book. If you were to write a book, would you skip hiring an editor? Of course not. The editor presents an objective third party. They're seeing the book for the first time. Similarly, a developer who is looking at a code review is seeing the code for the first time. This simulates what will happen when the code is running in the main branch of the repository — new developers will come across the code for the first time and have to make sense of it.

All in all, asynchronous is the way to go.

🚢 Good code reviews — Author expectations

@curtiseinsmann

Now when we talk about code reviews, there are two parties: the author, and the reviewer. Let’s talk about author expectations.

Good code reviews — Author expectations

- The PR should be small!
  - Easier to review
  - Less risk
  - Time effective for the author
- If it's too big, reviewers can reject

@curtiseinsmann

Reviewers can review faster. Reviewers can review throroughly. Reviewers can spend less time reviewing them. Many times the reviewer will need to look at other places in the code anyways.

Less risk. It's easier to find a defect or flaw when the code review is small, but when it's huge there's a problem. Also if there is a problem, it's easier to roll back.

Time effective — less wasted work if they're rejected. Easier to design a subset of a feature instead of an entire feature itself. Less blocking as well. Reviewers won't block the entire thing. Less conflicts was well!

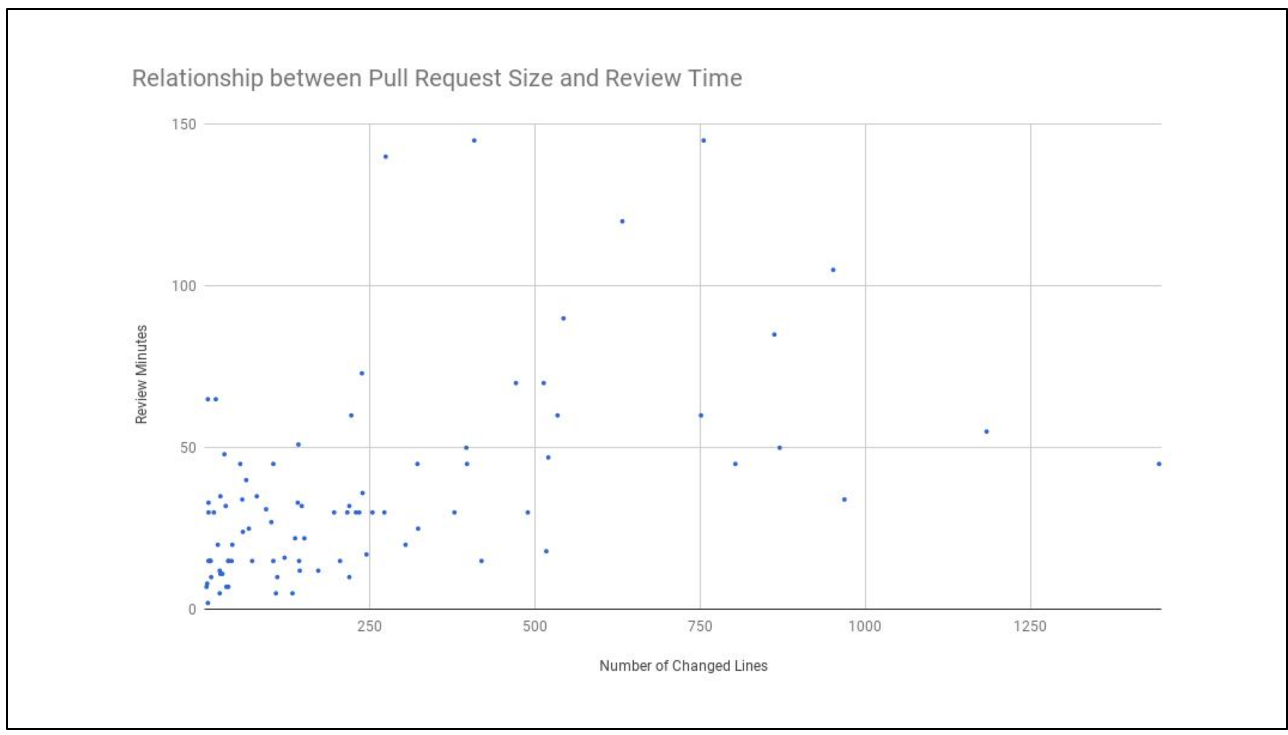


Diagram from: https://smallbusinessprogramming.com/optimal-pull-request-size/

So how small is too small?

No specific rules for this. This is just a guideline, maybe consider keeping it under 250 lines. But that is just estimation, and preferences vary. These should be decided upon by the team.

Good code reviews — Author expectations (continued)

- The PR should be scoped appropriately
  - Functional; includes tests
  - Passes builds
  - Can be deployed and rolled back safely
- Refactoring — ?

@curtiseinsmann

Should do one thing. You don't want a code review which fixes a bug with refunds, enhances the styling of a checkout button, and adds an entire table to the database. There should be a specific goal that the code review accomplishes, and a specific problem it solves.

It should be functional and include tests. I wouldn't recommend splitting implementation code and test code in different PRs. The code that's being added should be tested.

Passes builds and checks. It should be production ready.

Can be deployed and rolled back safely. When it goes out for code review, it should be in a state where it can be deployed. The change shouldn't be dependent on other changes, and if it is, this should be called out and merge should be blocked.

Refactoring varies by team. Some developers are OK with unrelated incremental refactoring. Some aren't. I'd recommend working with your team to establish guidelines on how this should be handled. In a separate PR, or the same one. This way the team is in sync about what's expected.

Good code reviews — Author expectations (continued)

- Include the *what* in the code review description.
- Include the *why* in the code review description.
- Extras if necessary
  - Related tickets, PRs, issues
  - Test coverage
  - Screen captures
  - Rollback safety
  - Backwards compatibility
- ***Assign*** the reviewer! (Prioritize familiar ones.)
- Drive resolution to conflicts — see it through to merge
- More in Part 3: Write better code

@curtiseinsmann

What should be Brief and concise
Code should be scoped and self explanatory

Why is more important than the what, when it comes to the PR description.
Why is the problem is worth solving?
Why did I choose to solve it this way?
Why did I choose the tradeoffs I did?

Extras if necessary. This will depend on the system you're contributing to.
<Enumerate> — Related tickets, PRs, issues. Test coverage. Screen captures. Rollback safety. Backwards compatibility.

Authors should assign the reviewer to the PR. In the Gumroad example, I'll show you how you can do this in GitHub. Prioritize the familiar reviewers — the ones who have last touched the code.

Drive resolutions to conflicts. If you're the author, it's your code review. You need to make sure you follow up and resolve discussions and disagreements, should they arise. Author shouldn't let the PR get blocked.

We'll go more into detail about how to be a great code review author in Part 3 — write better code.

Good code reviews — Author expectations (continued)

- Assign the ***right*** reviewer!
- Drive resolution to conflicts — see it through to merge
- More in Part 3: Write better code

@curtiseinsmann

Authors should assign the reviewer to the PR. In the Gumroad example, I'll show you how you can do this in GitHub. Prioritize the familiar reviewers — the ones who have last touched the code.

Drive resolutions to conflicts. If you're the author, it's your code review. You need to make sure you follow up and resolve discussions and disagreements, should they arise. Author shouldn't let the PR get blocked.

We'll go more into detail about how to be a great code review author in Part 3 — write better code.

🚢 Good code reviews — Reviewer expectations

@curtiseinsmann

Let's talk about Reviewer expectations now.

Good code reviews — Reviewer expectations

- Exhibit ownership and responsibility
- Get to them fast
- High standards
- Favor pragmatism, ***not*** perfectionism
- Be ***kind***
- Be ***thorough***
- Don't comment on style! Use linters.

@curtiseinsmann

Accept ownership and responsibility. If they're reviewing the code, it's as if they're writing the code themselves.

Get to the code review fast. Don't let it sit there for days and weeks. Within 24 or 48 hours is usually reasonable. Depends on your release cycle. For me, I prioritize code reviews in the morning, and after lunch. This unblocks my teammates in the morning and afternoon so they can work on addressing what I've said, or shipping what I've approved.

Favor pragmatism, not perfectionism. If it works, solves the problem, and improves the overall health of the codebase, then they should favor approving. We'll talk about blocking and unjustified blocking on the next slide.

Kindess creates an environment of empathic listening and mutual respect. If you learned something, tell them!

Review the whole change. Don't leave a review on lines 12 and 23, then on the next review leave comments on lines 34 and 45. This causes churn and discourages teammates. This takes time!!

Don't comment on style. Use linters or auto-formatters.

Good code reviews — Reviewer expectations (continued)

- Justified Blocking
  - Too big
  - Flaws, testing gaps, broken builds
  - Risks
  - Over-engineering
  - Obvious readability gaps
- Unjustified blocking
  - Nitpicks (especially style)
  - Emergencies
  - Utilize “comment and approve” when appropriate
- More in Part 2 — give better reviews.

@curtiseinsmann

Now let’s talk about when to reject. We do want our team to reject sometimes. We want functional, good code going through. But we also want our team to deliver quickly and on time.

Too big — remember we talked about scope, and the maximum number of lines our team should agree upon.

Flaws, testing gaps, broken builds

Risks — let’s say you’re approaching on a time of the year when you know your software is going to be serving a lot of traffic to one particular area of the application. Then you might want to hold off on modifying that portion until after the peak period is over.

Over-engineering. Unnecessary complexity that the team shouldn’t need to maintain.

Obvious readability gaps. If the code is too difficult to read and understand, and there are obvious simplification opportunities. The key word here is obvious. Remember that readability is subjective. So you can always make code more readable, but you don’t want to chase perfection.

Unjustified blocking:

Nitpicks. Things like renaming a variable or function. Things like a misworded,

one-line comment that only developers are going to see. As a team, you can define what these are for yourself.

Emergencies. This was the example I talked about before when the app was broken, and the ticket queue is growing to infinity. This is an emergent situation, and refactoring is not an option.

Utilize “comment and approve” — so for example if you want somebody to rename a variable, let them know that they should rename it, but also approve the PR, so they can address it before merge.

There will be more on how to give better reviews in Part 2 of this course.

🚢 Good code reviews — Guidelines

@curtisoinsmann

Of course, how do we get our team performing on all of the above? Best is guidelines. This is how you get your team on the same page, have them performing through these processes when you're not there.

Good code reviews — Guidelines

- Document in Notion or Team Wiki — collaborate with your team!
- Good guidelines have:
  - When and how to open a code review
  - Size and scope of PRs. Example: database migrations
  - Author and reviewer expectations
  - PR templates
  - Escalation procedures
  - When and when not to refactor
  - Team's stance on nuanced principles

@curtiseinsmann

You'll want these documented in your team Notion or Wiki. This shouldn't be an individual effort that you do by yourself. This should be a team collaboration so that you get on the same page. At least when you set up, then new team members can add input as they join.

Good guidelines. This will vary by team, depending on what you're building. For some teams, release notes are important. For others, versioning and backward compatibility is important. So be clear about what the PR is expected.

Here is what I think most teams should have:

When and how to open a code review. What does the flow look like? Should they open a draft first? How do they pick who to assign it to?

Size and scope. If there are database migrations, how should these be divided up compared to feature work?

Author and reviewer expectations. Most of these were documented in the previous slides.

PR templates. Maybe there are certain things you want to be required information in each code review that an author publishes. Like release notes and testing steps performed.

Escalation procedures. For example, when there is a disagreement, who do you reach out to in order to resolve the issue?

When to refactor. Some teams are OK with refactoring in the same PR. Some teams would rather it be pulled out into a separate PR. Come to an agreement on your team and stick with it.

Some teams value strong functions, some don’t. Some prefer duplication, some prefer DRY. These are opinion based and are worth discussing with the team and documenting.

@curtiseinsmann

OK! We've reached the end of talking about a good code review process. Let's recap.

Review early. Developers should be able to test their code on their own. Then they should create the review. The goal of the review is to ensure the code is production ready.

Set expectations for authors. Make code reviews small. Builds and tests should be passing. Include the what and the why and extras if necessary in code review description.

Set expectations for reviewers. Be kind and thorough. Document what is justified and unjustified blocking.

Document guidelines. Proactively resolve disagreements that may come up about refactoring and programming principles. Have a plan for escalation if there are disagreements.

In the next submodule, I'm going to walk you through an example code review process — Gumroad.

Example: Gumroad's code review process

@curtisseinsmann

What's up! In the last module we talked about the elements of a good code review process. In this submodule I'm going to walk you through an example of a good code review process.

I currently hold a part-time freelance software engineer position with Gumroad. It is a company known for its completely async culture with no meetings.

The Gumroad code review process is very similar to what I've experienced at Amazon as well. Except Amazon uses some private tools which may be foreign to most developers out there — but Gumroad uses many tools that are very much within reach of many development organizations. I'll just say up front that much of this was in place when I got here — Gumroad knows what they're doing in terms of documentation. I've contributed, but very little. I am able to recognize it as a good process though, because it aligns with what I've seen and built for my teams within Amazon. Plus, the company has a "build in public" culture, so I can actually show you a lot of what's going on behind the scenes here.

Let's start with the GitHub documentation. Within the repository itself there is a Shipping section which documents how to pick up work. As you can see it explains the entire flow here. Create a branch, then push to a remote branch, then assign a reviewer. So here is an example PR that I've already shipped. When I opened this, you can see that I was not working on the main branch called develop. I was working on a branch called filename. I'd pushed that to origin. Next I'd opened the PR as draft first — this is an option in GitHub where the PR can run checks, but it's not ready for

review. So then you can see by this check mark here, if I click into it, this is the automatic build that kicked off when I created a draft PR. If the build had failed, I wouldn't have opened it. If the build succeeded, I opened the PR for review. Then I assigned it, as you can see here. I assigned it to Ershad, because he'd touched the surrounding code recently. Notice how after they reviewed, they assigned it back to me. By using the assign feature, it's clear who needs to take the next step in the code review.

Notice how Gumroad is also using the Squash and Merge feature. So the individual commits don't get recorded. Each commit is tied to an approved PR.

Let's check on some other documentation that Gumroad has. Here in Notion they document pull requests in the engineering practices. Notice the specific call out of database migrations and how they want those to be handled. There are also code-specific ideas here. Using product instead of link in new code — they created a table called link to represent products, but that business domain name has changed.

They also have some specific guidance around how to use Git. Notice how rebase is destructive; they would rather you merge from develop into the feature branch you're using. This is a useful pattern to follow when you have a large number of engineers contributing to the same codebase — Gumroad has about 25.

Let's also take a look at some general principles documentation on GitHub. Duplication is better than the wrong abstraction. Leave things better than you found them — so this team is OK with incremental refactoring, and they're calling that out up front. This way the team is always on the same page. Notice that all these points are debatable — you may agree with them or disagree with them in general, but these are the opinions that the organization has come to, and they stick with them as a unit.

I want to also point out the Slack channel. There is this slack channel called #engineering where the PR comments and updates come through. This way, engineers are able to look at each other's code review in a timely manner.

Extras

- Buildkite: https://buildkite.com/
- Consider GitHub/Slack integration: https://slack.github.com/
- Amazon CodeGuru Reviewer: https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html

@curtiseinsmann

So that concludes the showing of Gumroad's code review process, used as a modeled example. If you want to check out GitHub/Slack integration, go to slack dot GitHub dot com.

Another useful tool is Amazon CodeGuru Reviewer, a public service with which I've used within the walls of AWS. It will automatically comment on your code for AWS best practices, concurrency risks, security analysis.

Key Performance Indicators — Forge a better CR process

- Decreased number of defects
- Decreased number of review cycles
- Speed of new developer onboarding
- Discussion quality

@curtiseinsmann

So that concludes the showing of Gumroad's code review process, used as a modeled example. If you want to check out GitHub/Slack integration, go to slack dot GitHub dot com.

Another useful tool is Amazon CodeGuru Reviewer, a public service with which I've used within the walls of AWS. It will automatically comment on your code for AWS best practices, concurrency risks, security analysis.

End of Part One! 🐂

Ok so congrats! You've reached the end of Part One! Give yourself some fist bumps! Remember that MJ is the greatest basketball player ever — with 6 championships, he was the ultimate leader — he knew how to motivate the teammates around him and put them in position for winning and success. After you've taken this module, you're now in position to put your team into a winning position with code review.

In the next module, we'll dive specifically into how to be a great code reviewer.

Part 2: Give better reviews

Drive code quality and elevate the skills of your peers.

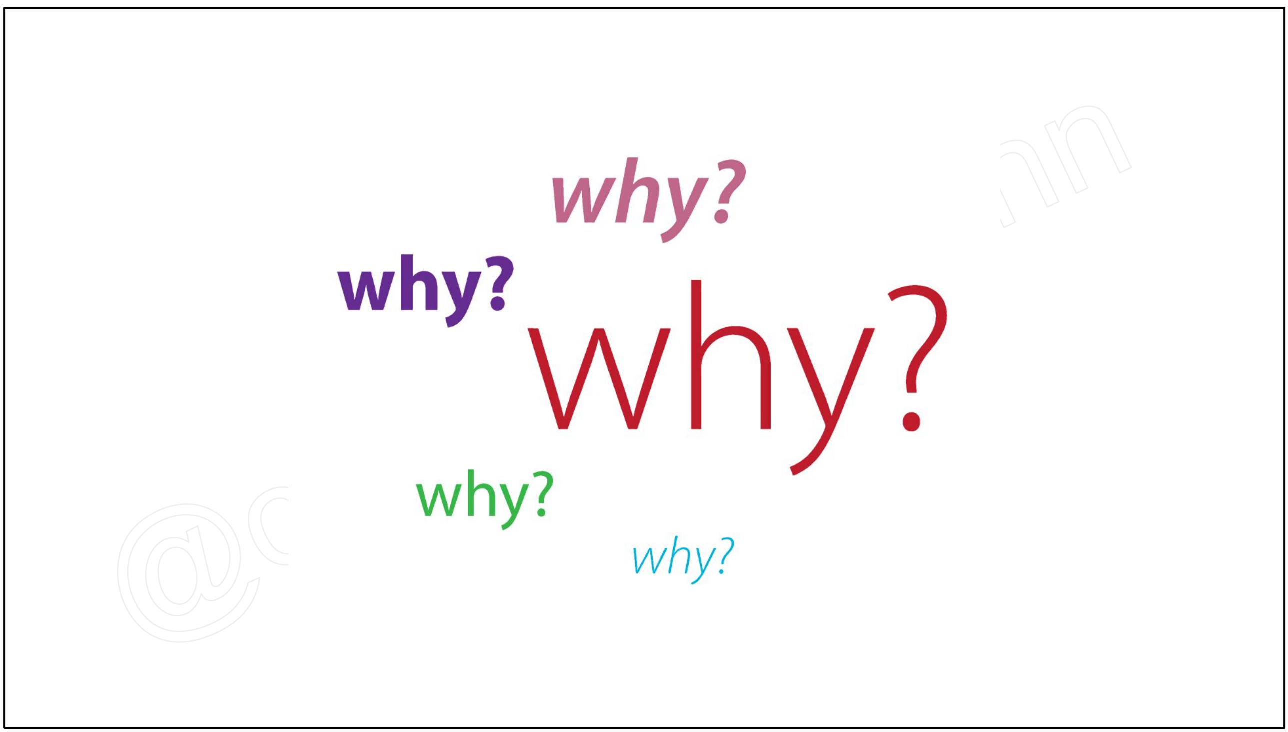
@curtiseinsmann

What's up! Welcome to Part 2 of the course — Master the Code Review! This part is called Part 2: Give better reviews.

In Part 1, we talked about forging a better code review process. We talked about how to set our team up for success by empowering them to ship faster.

It's good to set our team up for success. But as an engineer or developer on the team, we have to be an active participant in the success of code reviews as well.

This part will cover code review participation as a reviewer. Part 3 will cover code review participation as a code review author.



Why do we care about giving better reviews?

Earlier I showed you how Dropbox published their leveling guidelines. Let's take a look at the IC3 guidelines. This is your senior level at most companies, staff level at others.

First, this one:
"I ensure high code quality in code reviews."
A good reviewer knows what high code quality means. They know how to spot flaws and readability gaps, and they have a good process for doing so. They understand the line between perfectionism and pragmatism. They know how to raise the bar for code quality, while also building relationships with their peers. We're going to to talk about these elements in this part.

"I solicit and offer honest and constructive feedback that is delivered with empathy to help others learn and grow"
A good software engineer is seen as a coach and mentor. Code reviews are an excellent opportunity to cultivate those coaching and mentoring skills. We're going to talk about how to communicate insightful feedback to our peers.

"I tailor my message to my audience, presenting it clearly and concisely at the right altitude"
We will talk about how to tailor your message to your peer, depending on context such as who they are, what the team's goals are, and what they need to grow.

Why should I watch this module? (Agenda)

- What to look for in a code review 🔎
- How to perform a code review (step-by-step) ⚙️
- Writing effective code review comments
- Example review 👨‍💻

@curtiseinsmann

So, why should you watch this module? Here's the agenda:

We're going to cover what to look for in a code review. We're going to talk about the kinds of flaws we should be spotting. Before, during and after the reading the code itself.

We're going to dive into a step-by-step process for performing a code review. We'll unveil the mystery behind the intentional process that the best code reviewers use to find the flaws.

We're going to talk about how to write effective code review comments. We're going to improve our written communication skills to translate what we've found into actionable feedback for our peers.

And finally, I'm going to perform an example review.

With that, click into the next submodule, and let's get started with What to look for in a code review.

What to look for in a code review 🔎

@curtiseinsmann

What's up! Welcome to this submodule. In this submodule we'll talk about what to look for in a code review.

Our teammate has written some code. They've packaged it up into a Pull Request, Merge Request, whatever you want to call it. They've sent it to you to review. So what should you be looking for?

What to look for — why?

- Ownership and responsibility
- Flaws first
- Readability second
- Learning opportunities

Why are we talking about what to look for in a code review? When they send it to us, why can't we just approve it and let them ship?

Take ownership and responsibility. If you're reviewing some code, and you approve it, you're putting your name on it saying this was good to go. It's as if you wrote it yourself. When you write code, you don't want to produce bugs, because that code has your name on it. When you're reviewing code, you should have the same pride in the work. I'm not saying you're the person to blame if it goes wrong. This kind of distributed ownership works well for a blameless environment — you and your teammate(s) have collaborated. If something goes wrong in production, it's not a single person to blame — the team tried their best, that's all you can ask for.

Flaws first. We don't want flaws going to production. These can impact our customers that are using the software, and the business will suffer, and we'll add technical debt. We'll talk about some example flaws to look for. You are a line of defense for flaws getting out to the customers of your code. Even the best programmers make mistakes. Try to have a detective mindset.

Readability is secondary to flaws, but it's important. We read code many times more than we write it. If it's readable, we can add features faster, debug faster, and our life will be easier. We'll talk about specific readability points later.

Learning opportunities. Code review isn't always about teaching, it's about learning. Learn a part of the codebase you haven't dove into before, over time you'll get to

understand a lot of the system. Observe how the author solved problems, leverage these for the future.

Changes from all commits · File filter · Conversations · Jump to · 0 / 1 files viewed · Review changes

28 examples/lcp/Solution.java · Viewed

```
@@ -1,18 +1,22 @@
 1    import java.util.Arrays;              | 1    import java.util.Arrays;
                                           | 2  + import java.util.Objects;
 2                                         | 3
 3    public class Solution {              | 4    public class Solution {
 4  -   public String longestCommonPrefix(String[] strs) {   | 5  +   public String longestCommonPrefix(String[] strings) {
 5  -     Arrays.sort(strs);                 | 6  +     if (Objects.isNull(strings) || strings.length == 0)
 6  -     String first = strs[0];            | 7  +       return "";
 7  -     String last = strs[strs.length - 1]; | 8  +
 8  -     int c = 0;                         | 9  +     Arrays.sort(strings);
 9  -     while(c < first.length())          | 10 +     String first = strings[0];
10  -     {                                  | 11 +     String last = strings[strings.length - 1];
11  -       if (first.charAt(c) == last.charAt(c)) | 12 +     int position = 0;
12  -           c++;                         | 13 +     while(position < first.length()) {
13  -       else                             | 14 +       if (first.charAt(position) != last.charAt(position))
14  -           break;                       | 15 +         break;
                                           | 16 +
                                           | 17 +       position++;
15      }                                  | 18      }
16  -   return c == 0 ? "" : first.substring(0, c); | 19 +
                                           | 20 +     return position == 0 ? "" : first.substring(0, position);
17    }                                    | 21    }
18  }                                      | 22  }
```

“diff”

Briefly before we continue I want to define this term “diff” just in case people aren’t aware. I’ll use this word to represent a view like this of the change — you have red for the code that was deleted, green for code that was added, and so on.

What to look for — Before reading the diff

- Scope
- Builds, checks, tests
- Conflicts
- Screenshots
- Did they solve the problem — the right problem?

@curtiseinsmann

Now let's jump back into what to look for during a code review. So before you even read the diff, before reading those code changes, there are things you need to look for.

Remember in a future submodule we'll talk about how to specifically look for those things, but here we're focusing on what to look for.

You need to determine if the change is appropriately scoped. Most teams prefer small, incremental changes. Did they follow that guidance? If not, why? Is it OK? I'm usually OK with large changes if they're unavoidable, but most of the time they are.

Builds passing? In GitHub, there are ways to integrate builds and checks to make sure existing and new tests pass. You should verify that these are good to go. Testing— many teams have minimums for test coverage before pushing a change. Some don't. You need to make sure that the author has documented the manual tests they've performed, if necessary.

Conflicts. GitHub will automatically tell you if the branch has conflicts with the branch with the most updated code. If there are many conflicts, this might be an indicator that the author should merge first, because much of the code would need to be modified.

For UI changes, you may want to see screenshots or evidence that this is working locally.

Did they solve the problem? The right problem? This one is a bit tricker to detect, and it comes with experience. Remember that many problems can be solved without writing code. Many problems are not new, and other tools or libraries can be leveraged. Every line of code is something that your team will have to maintain going forward. The easiest code to maintain is the code that never gets written. Sure they went through the time investment to write the code already, but it’s a sunk cost if you have to maintain the code.

In a later submodule we’ll talk about a step-by-step process for how we’ll find these things. But these are the flaws we’re looking for.

What to look for — Flaws *within* the diff

- Edge cases / corner cases
- Testing
- Business requirement gaps
- Unexpected behavior changes
- Optimization
- Documentation
- Complexity / over engineering
- Style?

@curtiseinsmann

Now we'll talk about things to look for while you're reading the diff.

Missed edge cases and corner cases. Your null checks, your outliers, your off by one errors, your unhandled exceptional situations — the list goes on and on.

Testing. Does the code have unit tests? If your team uses integration tests, mutation tests, whatever they use — did they write them? Are there tests that should be added? Removed?

Business requirement gaps — remember to consider edge cases that the business hasn't given requirements for. For example, maybe you're building an some automation around external event notification system. The business has given you requirements about what to do when you see 2 event types, but the external system actually supports 4 types. Ideally you'd identify how to handle these in a design phase, before the code is written. But sometimes they surface in the implementation. Watch out for those.

Unexpected behavior changes. Let's say your API takes in an optional list of values as a parameter, but you would always accept null for this input. Then when modifying another list API, the API now throws an exception for a null parameter. That's an unexpected behavior change.

Optimization. Remember your data structures and algorithms — make a judgement on whether performance is important, and look for these appropriately.

Documentation. Different teams have different standards on documentation, make sure they're following for your team.

Complexity / over engineering. Remember, solutions should be as simple as possible so they can be maintained. If you see some complexity, consider if it should be there, or if there's an alternative solution.

Style? I put this one in question mark because there are automated tools that can format or check your style at commit time. Those should be leveraged. But there might be style problems you come across that you need to add to those rules.

What to look for — Flaws *outside of* the diff

- Partial refactoring
- Side effects
- Changes that aren't backwards compatible
- Rollback risks

@curtiseinsmann

OK we've talked about flaws within the diff. Now let's talk about flaws outside the diff. These are the types of flaws that may not be visible or obvious by reading the diff itself. Ideally these would be caught by tests, but it doesn't always happen.

Partial refactoring. There was some refactoring done, but wasn't considered in enough places. For example, the signature of a function is changed, as well as its usage in one place, but that function is actually being called in 3 other places and those callers need to be changed too.

Side effects. Let's say you add a new status type or enum to a database record type, but you have async workers that process the records and assume certain status types. Those workers might run into an exception.

Changes that aren't backwards compatible. When you push out new versions of your software, the things that are integrating with the older versions of your software should still work. If they don't, the change is not backwards compatible. Let's say you're pushing an update to your API that introduces a new parameter, and that parameter is required. This is not a backwards compatible change, because the clients of that API aren't using that required parameter, and they'll break.

Rollback risks. If you deploy the software, and then roll back, there might be breakage. For example let's say you're queueing up messages to be processed in a background job. The background job expects a certain format of the message, and validates it. Let's say you push out a change which changes the message format, but

also update the validation on the worker side. If you were to rollback that change, your workers may still receive background job messages from that changed format — will they fail?

What to look for — Readability gaps

- Intent should be obvious
- Naming
- Abstraction
- Directories
- Conventions
- Implementation *and* tests

@curtiseinsmann

Now that we've talked about flaws, let's talk about readability gaps. We read code many times more than we write it. If more readable our code, the easier it'll be for our team to debug and add features.

Most of these are subjective. A full enumeration of code readability principles is out of scope for this course. Here I'll call out some of the most important ones.

Intent should be obvious. You should understand what the code is doing. If you're good enough to get a developer job, you're good enough to understand the code on your own. If there is some sort of obscure workaround or hack or complexity, it should be obvious as to "why" it needed to be performed.

Naming. Variable names should be concise, and descriptive about what they represent. Making sure that function names say what they do. Names should match the business domain — if there's a difference between a purchase and an order, that should be reflected in the code.

Abstraction. If you have a class which is intended to interact with the AWS S3 API, it shouldn't be making calls to AWS DynamoDB.

Directories. Making sure classes and files are placed where they're supposed to be. If you have a folder for your high level API handlers, don't put a general utility class in that same directory — it should be with other utility classes.

Conventions. If you're writing JavaScript or Typescript and the entire file they're modifying is using promises, but the author's change is written in async/await, this is something to call out.

Implementation and tests. Hold the same high standard for readability. For example in Ruby tests, there are two ways you can organize groups of individual tests — with the describe keyword, or the context keyword. The change should be consistent with what's already there.

[These are not affiliated links]:
Clean Code:
https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/B08X8ZXT15/ref=sr_1_1?crid=O3FQ0FKG9612&keywords=clean+code+a+handbook+of+agile+software+craftsmanship&qid=1641150785&sprefix=Clean+Code%2Caps%2C386&sr=8-1

Code Complete:
https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670/ref=sr_1_1?keywords=code+complete&qid=1641150820&sr=8-1

A Philosophy of Software Design:
https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201/ref=sr_1_1?keywords=a+philosophy+of+software+design&qid=1641150857&sprefix=a+phi%2Caps%2C356&sr=8-1

Recap — What to look for in a code review

- Take ownership
- Flaws
  - Before the diff
  - Within the diff
  - Outside the diff
- Readability gaps

@curtiseinsmann

OK let's recap! What to look for in a code review.

Have ownership. You're the first line of defense against flaws going to production, and you'll be maintaining the code. Have high standards, use it as an opportunity to raise code quality and learn from others.

Look for flaws. This is the highest priority. You don't want flaws going to production. Humans make mistakes, even the best programmers.

Flaws before the diff. Is the change scoped properly? Are the checks, builds and tests passing? Screenshots, should they be included? Did they solve the right problem?

Flaws within the diff. Edge cases, corner cases. Gaps in the business requirements. Optimization — does it need to be optimized, and are there performance improvement opportunities? Does it follow documentation standards set by your team? Is there any unnecessary complexity or over engineering?

Flaws outside the diff. Partial refactoring. Side effects. Backwards compatibility. Rollback safety. Thinking about the context of the change in the system as a whole.

Readability gaps. Obvious intent — you should be able to read it and understand it. Naming. Abstraction. Following conventions.

Now that we've talked about what to look for in a code review, the next section will go

through a methodical, intentional process for performing code reviews.

Now that we've talked about what to look for in a code review, let's talk about a step-by-step, methodical, intentional process that we can use in order to perform that code review and find what we're supposed to.

Main article: https://curtiseinsmann.medium.com/ive-code-reviewed-over-750-pull-requests-at-amazon-here-s-my-exact-thought-process-cec7c942a3a4

How to perform a code review — why?

- Avoid analysis paralysis
- Provide a transparent thought process
- Give good feedback

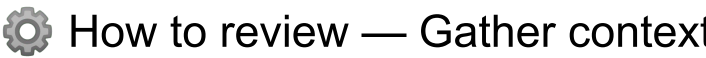
tiseinsmann

This section will get pretty granular. So why am I talking specifically about how to perform a code review?

First I want to help you avoid analysis paralysis. Reviewing code can be overwhelming. Every line of change added or removed could cause something to break. We want to have an intentional strategy to make sure we're catching what we're supposed to catch, allowing what we're supposed to allow, so that the author can deliver.

Provide a transparent thought process. When I first started my career, I would author a code review and send it to a teammate. Then, comments would appear. But I didn't understand how they got there in the first place — I didn't understand the step-by-step process that my teammate was following. Early code reviewers tend to gloss over with their eyes when the code is in an internet browser. This section will help you with that.

Give good feedback. In the section after this one, we'll talk about writing effective code review comments. We want to follow a process that will set us up to give accurate, unambiguous feedback.



- Read the code review description, and related issues/tickets
- Determine if you're the appropriate reviewer
- Set aside time
- Optimizing for speed or quality?

@curtiseinsmann

How to review. First you need to gather context.

This first point is a bit obvious, read the code review title and description. Read the issues that the code review resolves. I see reviewers make the mistake of diving into the code too quickly. Code is very difficult to understand if you don't know what it's supposed to be doing.

Determine if you're the appropriate reviewer. Ideally it should be somebody familiar with that part of the system, or somebody that recently touched the surrounding code. If you're not the right person, it might be an opportunity to assign it to someone else. There are situations where you may be the only person on your team who can review. Or maybe your team has had a lot of turnover. In these situations you may be forced to review something you're unfamiliar with — just know that this will take more time, and communicate it with your managers and stakeholder.

Speaking of time — you need to set aside time to perform this code review. Code reviews take time. I've spent 5 minutes, I've spent 2 hours. It really depends on the nature of the change. These should be factored into your estimations and bandwidth for how much work you can get done in the week. It should influence how many tasks your taking on.

Next you need to understand whether the author is optimizing for speed or quality. Are they working against a tight deadline? Is the change urgent? Is there time to focus on quality? Considering this will give you the context on how high your standards should

be.

How to review — Read for understanding

- Start with the crux
- Read randomly 🧩
- Don't forget!
  - Context lines
  - Existing repository code
- Go class by class after you understand
- Flaws — you're a detective

@curtiseinsmann

After you've gathered context, read for understanding.

Start with the crux. When you start to look at a code review, there are often changes to many different files. The crux is where the focus of the change is. It's the critical part. When you're talking about the diff, it's usually where you see a 50 lines of green, or 50 lines of red. It may take you a couple of scans to understand where this is. Examples are the API entry point or the refactoring of the data access layer.

Next, read randomly. When's the last time you coded something once and it was good to go? Never. You had to write and rewrite it. Same thing for reviewing a CR. You'll have to read and read and read. You may have to read the same code twice even three times to understand what it's doing. Every line of code is a puzzle piece, and you slowly start to piece together the puzzle.

As you're reading, don't forget:

- context lines. On almost every code review, I'm expanding context lines. You'll very rarely see a code change that doesn't affect the surrounding code in some way. It's not necessary to expand all context lines, but just enough so that I understand what the change is doing.
- Existing repository code. Again, it's not necessary to read the whole codebase. But think about the components that the change is integrating with. You may need to go and read the boundary code for those in order to understand the change. This is also how you spot your side effects that we talked about in the "What to look for" submodule.

Once you understand what the change is doing, go class by class. Or function by function. This is a good chance to perform a side-by-side comparison of the class and its unit tests. You read the class, and think about what unit tests you expect to exist. If they don't, it's an opportunity to call it out.

Flaws — you're a detective. This is a mindset thing. Pretend like you're a detective, and that there's a flaw lurking and hiding in the code somewhere. This will open your eyes to catch these types of things.

How to review — Commenting

- Write comments as you go — but they'll be wrong
- You will need to rewrite your comments
- Leave positive comments — don't over do it
- Make a clear approval decision

@curtiseinsmann

We'll go over effective and ineffective comments in the next section, but there are specific things to call out here.

Write comments as you go piece together the puzzle. As you spot an edge case, as you come across something confusing. But, these comments will likely be wrong, just because you haven't finished the puzzle yet. For example let's say you see a missing null check in a function. But then after seeing the caller of that function, you realize that this check was already performed. So save them as a draft comment first, and publish your comments at the end.

You will have to rewrite your comments. After you piece together the puzzle, you'll have a clearer picture of what the change is doing. So that's when you go back and revise your comments. Then, send them to the reviewer all at once. We'll talk about what comments should look like in the next module.

Make a clear approval decision. Remember in part 1 we talked about justified blocking and unjustified blocking. If there's a flaw or risk, don't approve the code. If you only have minor comments or nitpicks, approve it. Be pragmatic, not a perfectionist. The most important part is does it work, and is it readable enough to ship.

Recap — How to perform a code review

- Gather context
- Read for understanding
- Write comments as you go
- Make a clear approval decision

@curtiseinsmann

Writing effective code review comments 📝

@curtisoinsmann

Let's talk about writing effective code review comments. When you're performing a code review, there are two dimensions. You're reading the code, trying to find flaws and readability gaps.
But then you need to communicate with your teammate, in writing, about how to address what you've found.

Effective code review comments — why?

- Reduce churn
- Teammates will respect your opinion
- Example artifacts of good work

Heinsmann

So, why do we care about writing effective comments?

We want to reduce churn. Churn comes from ambiguity or inaccuracy. If you leave an ambiguous comment, you could cause a lot of back and forth, or you could cause your teammate to go down the wrong direction with implementation. Both of these could cause shipping delays for the author.

Teammates will respect your opinion. If you're kind and insightful, you'll build relationships and credibility to have input in other areas.

Example artifacts of good work. If you want to level up, you need examples of mentoring and coaching. Code review comments are a great way to do this.

Effective comments — The code, not the person

❌ "You didn't check for a null value."

✅ "This input value could be null, causing a server error. If null, a client error should be thrown."

OR

✅ "What would happen if the input value is null?"

@curtiseinsmann

Let's start with an example. I've seen this kind of comment many times in code reviews.
I don't like this because they use the words you. I don't think you should ever use that pronoun in code. In doing this the reviewer is commenting on the person. The target for the comment should be the code.

This input value… This example is better. It targets the code. It also gives a reason why a null problem is bad — the server error could be caused. It also gives a suggested path forward. This way the author knows what to do.

The other alternative that I like is this one. "What would happen…" This gives the author a chance to find the solution themselves. The tradeoff here is they may not havea path forward. Use this if you have some time until delivery, or you're trying to grow a junior so they can solve the problem on their own.

[Just a note] Good example:
https://twitter.com/curtiseinsmann/status/1386548342159597569?s=20

Effective comments — Nitpicks

✅ "**Nit, non-blocker**: I'd recommend changing the variable name from 'purchaseStatus' to 'orderStatus'. The variable stores the status of an order, not a purchase."

@curtiseinsmann

Next example, let's talk about nitpicking. I absolutely believe that you should nitpick. Minor improvements to a codebase compound as it gets larger. However, you should do it in the right way.

This first example is an unproductive nitpick comment. It sounds like a command, like you're ordering them on what to do. This is detrimental to relationships. Plus in most situations, you shouldn't be blocking somebody's code review from being pushed because of a change in variable name. It's just too minor.

This example is better. It's prefixed with nit — this disarms the individual from getting defensive, they know it's minor. It's very clear that it's a non-blocker. If this is the only comment on the code review that you have, you should approve it, and still leave this comment. Trust that the author will address it before merge. It also gives a reason for why the variable name is bad. Purchase and order likely mean different things in the business domain. And this is an important change.

Effective comments — Have a reason why

❌ "Query the table index, not a GSI."

✅ "This queries a Global Secondary Index, which will be an eventually consistent read. There are race condition situations where the record's version could get updated by another handler immediately before this query. This will fail optimistic locking and throw an exception. This needs a strongly consistent read — the DynamoDB table index should be queried here."

@curtisoinsmann

This example gets a little more technical — it's about DynamoDB, which is an AWS service. But don't worry if you've never worked with it before.

In this example, the author is performing a query on the wrong index.

Now this comment says "Query the table index, not the GSI." Leaving it here is a bit ambiguous — it's not clear why the table index should be queried. The term GSI may be unfamiliar as well. The author could make the change, but they wouldn't understand why they did it.

This second comment goes into more detail. And it's a big one, and I'll go through it piece by piece, so bear with me.

Notice how each sentence builds upon itself — one sentence is written, then the next sentence answers "why is that bad?"

It expands the acronym (Global Secondary Index). It talks about the effects of querying on the GSI — an eventually consistent read. It talks about why is that bad — the version could get updated in a race condition. Why is that bad — it'll fail optimistic locking and throw an exception.

Then the final sentence suggests the path forward — a strongly consistent read.

So this takes a long time, yes. 2 reasons for this.

1. The process of writing will clarify your thoughts. There are times when I’ve written a few sentences of a comment, then realized I was wrong, or had come to the wrong conclusion. This will help you become more accurate in comments.
2. It’s a log term investment in the junior engineer. Reading the first comment will probably make the change, but they won’t understand why, and they may be prone to the same mistake in the future. These reasons why solidify the concept in their brain. Remember, code reviews are an excellent opportunity for coaching.

One final thought on this slide. Remember to keep your audience in mind when writing out comments like this. Let’s say the code review author is a senior engineer. Or maybe it’s somebody who already knows DynamoDB well, they just made a simple mistake. It might not be necessary to explain all of this. Remember to keep in mind about who your audience is.

Effective comments — path forward

❌ This solution is not optimal. The time complexity is O(n^2)

✅ The time complexity here is O(n^2). If we sort the array first, we can achieve a time complexity of O(n*log(n)). This will be processing a large data set in a synchronous critical path, so it's worth optimization.

@curtisoinsmann

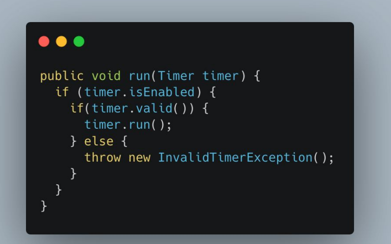
This example is similar to the previous ones, but adds in a flavor of performance.

This comment says "This solution is not optimal. The time complexity is O(n^2)." This does well by talking about the code, not the person. But it doesn't give a reason why it's bad. Sure, time complexity is O(n^2) — but this is OK in many situations. Performance seems objective, but it's often subjective — opinions on what performance is acceptable and what isn't.

This second comment is better. It gives a path forward. It gives a reason why optimization is worth it — we don't have context about the system and code, but perhaps it's being run on a synchronous an API invocation, where the server needs to respond to the client. Exact numbers would be useful in such a comment here.

[Notes only]
You need to give them a path forward. Here is an example of a solution where the person gave an ambiguous comment. They didn't give the person a path forward in how they can correct their approach.
https://leetcode.com/problems/two-sum/discuss/942481/JAVA-Solution-or-OPIMAL-or-100-RUNTIME (this is leetcode, so not exactly a code review. But if I frame it like a code review, then I can make my point better.)

Effective comments — background



✅ "Nit: I'd recommend a guard clause here. This makes the special cases immediately obvious to future readers. It saves them cognitive space while reading the rest of the function. For more context, see: https://medium.com/@scadge/if-statements-design-guard-clauses-might-be-all-you-need-67219a1a981a"

In this example I did want to actually put some code here. So in this code we have a lot of if/else statements and branching.

In the comment the reviewer wants the author to use an early return. It's a nit, so they've correctly prefixed with nit. But two bad things — one it sounds like a command, telling someone what to do — you could alienate your teammates like this. Two, they may not be familiar with what a guard clause is, or why it matters.

So in this second example the reviewer is providing a recommendation. They are also providing a reason why, and linking to an article. The reason why is going above and beyond here — sometimes it's acceptable to just post the article.

Article:
https://medium.com/@scadge/if-statements-design-guard-clauses-might-be-all-you-need-67219a1a981a

```java
public void run(Timer timer) {
  if (timer.isEnabled) {
    if(timer.valid()) {
      timer.run();
    } else {
      throw new InvalidTimerException();
    }
  }
}
```

```java
public void run(Timer timer) {
  if(!timer.isEnabled())
    return;

  if(!timer.valid())
    throw new InvalidTimerException();

  timer.run();
}
```

And just so you know what the code would look like after creating a guard clause. On the left you have the original code. On the right you have the same function rewritten with guard clauses. This has a minimal advantage with a small function like this. But imagine if the function was 10, 50 lines long and had several different branches — readability would certainly increase there.

Recap — Writing effective code review comments

- Comment the code, not the person
- Propose a path forward, or collaborate
- Use “nit” and “non-blocker”
- Give a reason why
- Provide background

@curtiseinsmann

OK, let’s recap!

Go through bullets…

Now in the next section, I’m going to perform an example review.

Example code review 👨‍💻

@curtiseinsmann

What's up! Welcome to this submodule. In this submodule I'm going to perform an example code review, and incorporate some of the things we've learned from previous submodules — what to look for, how to review, and writing effective comments.

We're going to take a look at some example code — but I want you to review it on your own first, before you see the comments I wrote.

Now in a real GitHub code review setting, we'd be looking at a diff, where the change is part of a larger repository that we're familiar with it. Well, I don't have a shared repository for you. Even if I did, I'd have to give you time to understand it, to simulate a code review with the paradigm of familiarity.

So we'll get creative here. We'll have a function of example code, with some context about what it's supposed to do. Then we'll review it. But I want you to pause the video, and review it yourself. It's the only thing I'm asking you to do on your own in this course, so I expect you to do it! When I go to the next slide, it'll have the code — you don't need to pause until I tell you to. I'll walk you through the setup first.

@_rodrigomd

@curtisreinsmann

```
async function subscriptionsProcessor(event, context) {
  let transactionId = context.transactionId;

  try {
    if(isUpdateEvent(event)) {
      await processUpdateEvent(event, context);
    } else {
      await cacheMappingReference(event.payload, context);
    }

    await cacheEntry(context, event.payload);

    if (isUpdateEvent(event) && context.products &&
          Object.keys(context.products).length < 0) {
      return;
    }

    await transformAndPublishSNSEvent(context, event);
  } catch (e) {
    log.error({
      errorMsg: `Error processing event: ${err.message}`,
      err: err,
      transactionId
    });
    throw err;
  }
}
```

```
The subscription-processor is invoked every time a
subscription event is received.

The event could be a subscription create or update.

The processor is in charge of:
- caching the event payload
- caching a mapping entry that will
  be used for legacy systems
- transforming the received event
  into SNS event format
- publishing an SNS event EXCEPT:
  - when it's an update event, AND
  - the context contains an *empty* "product" object
- logging any error
```

OK, so here's the example. You don't have to pause yet, I'll tell you when.

This example comes from Twitter. Rodrigo, another outstanding engineer I follow (you all should go follow him), created a Tweet asking for people to review the code.

https://twitter.com/_rodrigomd/status/1458389170645204993?s=20

The code I've shown here isn't exactly what he posted. So I've modified it a bit for clarity here. The requirements are on the right, the code is on the left — only because my video is on the right and this is how it would fit.

A few clarifications about the code:
SNS is an acronym for Simple Notification Service, which is an AWS service, but that's not so important here.
You'll notice that this code is calling other functions. You can assume those functions exist and do what they're supposed to do.
You can assume that log is defined somewhere in the application.
I haven't included tests for the code, you can assume it's covered by tests.

I've already reviewed the code and added comments. I'll walk through my review process, and the comments I came to on the next slide.

When you review this code, you and I will likely catch different things, and have different comments. That's OK. Reviewing code is like an art form.

What I want you to do now: read the description on the right side, then read through the code on the left side, and write some comments. I've already done this. I'll share my review process and I'll share my comments on the next slide.

At this point, feel free to pause the video and review the code.

OK, welcome back! How'd you do? Let's continue on to the next slide.

The subscription-processor is invoked every time a subscription event is received.

The event could be a subscription create or update.

The processor is in charge of:
- caching the event payload
- caching a mapping entry that will
  be used for legacy systems
- transforming the received event
  into SNS event format
- publishing an SNS event EXCEPT:
  - when it's an update event, AND
  - the context contains an *empty* "product" object
- logging any error

```
// Inspo: https://twitter.com/_rodrigomd/status/1458389170645204993
async function subscriptionsProcessor(event, context) {
  let transactionId = context.transactionId;

  try {
        if(isUpdateEvent(event)) {
        await processUpdateEvent(event, context);
        } else {
        await cacheMappingReference(event.payload, context);
        }

        await cacheEntry(context, event.payload);

        if (isUpdateEvent(event) && context.products &&
        Object.keys(context.products).length < 0) {
        return;
```

```
        }

        await transformAndPublishSNSEvent(context, event);
  } catch (e) {
        log.error({
        errorMsg: `Error processing event: ${err.message}`,
        err: err,
        transactionId
        });
        throw err;
  }
}
```

```
async function subscriptionsProcessor(event, context) {
  let transactionId = context.transactionId;

  try {
    if(isUpdateEvent(event)) {
      await processUpdateEvent(event, context);
    } else {
      await cacheMappingReference(event.payload, context);
    }

    await cacheEntry(context, event.payload);

    if (isUpdateEvent(event) && context.products &&
          Object.keys(context.products).length < 0) {
      return;
    }

    await transformAndPublishSNSEvent(context, event);
  } catch (e) {
    log.error({
      errorMsg: `Error processing event: ${err.message}`,
      err: err,
      transactionId
    });
    throw err;
  }
}
```

💬 Nit: transactionId is only used once, and not until the catch block. The declaration here causes the reader to expect it'll be used in the critical path. I'd recommend eliminating this variable.

💬 The event's structure should be validated, since it's received from another system.

💬 According to the PR description, the event payload and mapping entry should be cached every time. Shouldn't both of these happen at the beginning?

💬 Shouldn't the mapping reference still be cached even if `isUpdateEvent` evaluates to true?

💬 `err` should be `e`

I'm going to walk you through my review process first. Then I will show you the comments I arrived to.

So I'm reading through this code. I understand that it receives subscription events and processes them. If I were reviewing this in a repository setting, I'd probably follow these function calls and give them a onceover, just to make sure they do what they say they do.

Now I'm looking at this transactionId. I'm trying to figure out where it's being used. But I see that it's only used in the catch block. And this leads to our first comment. Nit… eliminating this variable. Notice how it's a nit, because it's a simple elimination of a variable. I'm giving a reason why. And I'm also recommending a solution.

I'm thinking about how this event is coming in. I notice that we are doing some processing on it immediately. But thinking about where the event is coming from. It's coming from an external system. We need to verify that it's what we expect before processing it. It's true that maybe these helper functions might be doing their own validation. But it's not something I see here. So that leads to the next comment. The event's structure… another system. This is a concise comment, and it gives a reason why. I could expand here, depending on the level of the author.

Now I'm reading through the code, and I notice that things are happening in a certain order. And I compare them with the description of what it said. So if we go back to the description, it reads like the caching of the payload and the mapping reference should

happen every time. So it leads me to this comment. According to… beginning? And I ask this as a question because really, I'm not sure what should be happening. I don't know if the code is right, or the description.

Then I notice that the caching of the mapping reference will only happen if this evaluates to false. So my next comment, also a question. Shouldnt we…. true?

Now I'm reading through the error handling. I notice that the err should be e. Error handling code in particular is something you should watch out for. It often goes untested and many bugs are caused by error handling because the code is rarely run in production.

So I'm reflecting on the entire change. I like to ask myself the question, what would happen if something goes wrong? How could we diagnose this? That's how I come to this logging comment. Another thing to think about is instrumentation — do we have metrics around the types of events coming in? If not, it might be good to add those.

And overall, I would not approve this code. It has a couple minor readability gaps, but it has flaws. So it is definitely a blocker. But notice how with these comments, the reviewer will be able to address them. And we should be able to get this shipped by the next iteration of the review, provided that the answers to the questions get resolved about ordering.

Now these are the comments I came up with. It's OK if you came up with different ones, worded them differently, etc.

Key Performance Indicators — Give better reviews

- Peers aren’t repeating the same mistakes
- Peers are shipping in less reviews over time
- Teammates are repeating similar comments
- Qualitative feedback from mentors/managers

@curtiseinsmann

So that concludes the showing of an example code review. And we’re approaching the end of this entire module, give better reviews. So the question is, how do we know if we’ve given better reviews?

So yo

End of Part Two! 🐂

Congrats for getting through Part 2 — give better reviews!

Remember, MJ was the greatest basketball player ever. During practices he pushed his teammates to get better. He was able to bring out the best in others, because he knew he needed to count on them when they were called on the court. He made the people around him better.

In this module you've learned the skills you need to make the people around you better and help them level up their skills. This will put your team in position to win.

Now the next module will talk about how to write code that gets approved quickly in a review, so that you yourself in a position to succeed at code reviews and ship fast.

Part 3: Write better code

Craft beautiful code that gets approved in the first review.

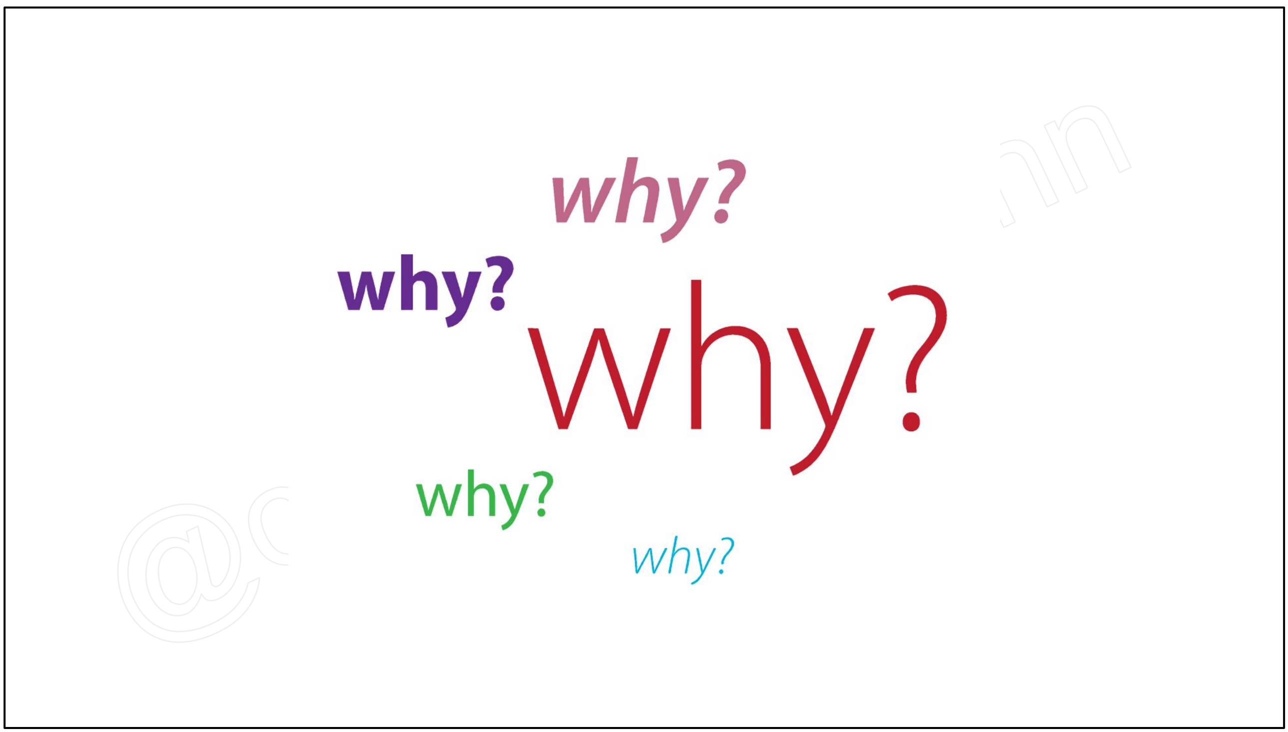
@curtiseinsmann

What's up! Welcome to the 3rd and final module of Master the Code Review. Part 3: Write better code. Craft beautiful code that gets approved in the first review. We're going to dive into code reviews from the author side.

Now I originally intended for this to be the first module of the course. However, I realized that you would be able to follow the advice in this course, but if you're in a dysfunctional code review environment, or you're dealing with a bad reviewer, shipping in the first review, or minimal reviews is out of your control.

So that's why this module is last. In the first module we established the environment and laid out a good code review process. In the second module we talked about one participant in that process — the reviewer. With that knowledge, going forward as you watch this module, you'll better be able to identify if it's your code that needs improving, or your environment.

But provided you have a decent environment, and decent reviewers, this module will show you how you can be successful as a code review author.



Why do we care about writing better code? Why get it approved in the first review?

Earlier I showed you how Dropbox published their leveling guidelines. Let's take a look at the IC2 guidelines. This is your mid-level at most companies, senior level at others.

First, this one:
"I translate ideas into clear code, written to be read as well as executed."
Written to be read is the important part of this sentence. We read code many times more than we write it. The code should work, but it should also be logical and readable.

"I independently define the right solutions or use |existing approaches to solve defined problems."
Key word here is independence. If you're a junior engineer or early career developer, one of the key aspects is to complete tasks on your own. It's OK to seek guidance and help when you need it. But you'll find that you can independently solve most problems if you just stick to it and learn how to operate independently.

"I listen to understand others and ask clarifying questions"
We will inevitably receive feedback when we write code. That's OK. The important piece is to learn how to listen to this feedback, take it in stride, and ship the best code we can produce for our team.

One thing that isn't on here. Many companies track how many reviews it takes you to ship code. Of course it needs to be taken in the context of how good and how strict the reviewers are. But the less number of reviews, the better. What this means is that if you can get it done in one review, get it done in one review.

Why should I watch this module? (Agenda — Write better code)

- Principles 📕
- Process (step-by-step) ⚙️
- Addressing comments 💬
- Example code reviews 🍬

@curtiseinsmann

So why should you watch this module? Here's the agenda.

First I am going to teach you some principles to write better code. We're going to look at several code examples and talk about the readability.

You will learn the process to write better code from the time you get a task to the time you open it for code review. It is a step-by-step walkthrough, similar to the step-by-step walkthrough I gave in give better reviews.

How to address feedback. There will be feedback, there will be comments, there will be discussion. You need to handle this so that you ship and improve long term, shipping in fewer iterations.

We're going to walk through some example code reviews that I myself have authored. I'm going to walk you through the task and how I got the code approved.

With that, go ahead and click into the next module. Let's start with principles to write better code.

Principles to write better code 📕

@curtisseinsmann

Hey! Welcome to the first submodule of Write better code — Principles to write better code. In this submodule we'll talk about some principles that will help you write readable code, and we'll go through several side-by-side code examples.

Principles to write better code — why?

- “**Craft *beautiful*** code”
- Reduce number of reviews to ship
- Some companies track number of reviews

Why are we talking about these?

Now if you recall, this high level module is called write better code, craft beautiful code. These words are very much intentional.
Coding is a craft. It's a form of logic-based creativity. Every task, every new feature is a blank canvas. It's an opportunity to demonstrate your abilities and produce something beautiful.
Beautiful is subjective. What's beautiful to me, likely won't be to you. You may even think my code is the antithesis of beauty, and that's OK.
You'll have to decide what beautiful means for yourself. It'll change over time. For now, decide that each code review you open will be something you're proud of. Every time.

Reduce number of reviews it takes you to ship. Obviously when you work hard on the code you write, you want to deliver it as soon as you can. Code reviews can be a blocker to that. The best way to get past the review stage is to produce code that works, and code that’s readable. If not, you could cause review churn, which results in delayed delivery and distractions for your teammates.

Also keep in mind that there are some companies that track the number of reviews it takes you to ship your authored code. Now this quantitative metric can’t be used without qualitative context. Such as how strict the reviewers are, the complexity of the system you’re working on. But you want to drive down the number of aggregate reviews over time.

Principles to write better code — The Main Questions

- Does it work?
  - ***OBJECTIVE***
- Is it readable and maintainable?
  - ***SUBJECTIVE — based on opinions!***

@curtiseinsmann

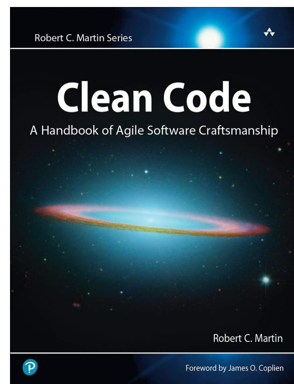
Now when a reviewer looks at your code they're going to be looking at two things.

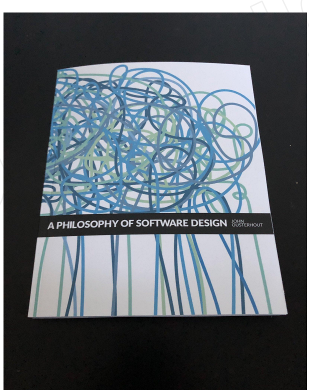
Number 1: Does it work? Does it solve the customer problem? Does it handle edge cases and corner cases? Is it tested and performant? Does it use the write data structures and algorithms?
Remember that this is pretty much objective. Your code needs to work. If you're hired as a developer I'm going to assume you know data structures and algorithms and how to handle edge cases and corner cases. Spending time on this subject wouldn't be productive in the scope of this course.

Is it readable and maintainable — now there are books written about this. You have Clean Code, Code Complete, A Philosophy of Software Design. But the catch is, all of these are subjective, and based on opinions! [Next slide]

Principles to write better code — Readability is *subjective*

- Reviewers shuffle
- “Functions should be small”





So let’s think about that a bit, readability is subjective.

Reviewers shuffle. You will have different teammates all the time. You would have one teammate reviewing a feature one day, another reviewer has different opinion on it. You can have one reviewer on your team approve something, another one wouldn’t. You can switch to a team which is really strict, and one that’s not.

“Functions should be small.” Bob Martin, author of Clean Code says functions should be small. Dr. Ousterhout (excuse me if I’m butchering the name) from A Philosophy of Software Design says functions shouldn’t be divided up because it leads to bad abstractions.

So who’s right? Both, and neither. It’ll depend on your preferences. I used to agree with Bob Martin, now I agree with Dr. Ousterhout. But that could change in a few weeks or years.

But remember, code reviews are based on these opinions, and there are many conflicting opinions. But we want to succeed in code reviews. Which leads us to the question…

*How do we write readable code which **gets approved quickly**, in a **code review** environment of **conflicting opinions**?*

@curtiseinsmann

<Say question>

I can’t give you a straight answer to this. Remember I’ve had 90% of my code reviews approved in the first revision at my last year at AWS. So I’ve had some success with this, and I can tell you what’s worked for me.

So here’s what’s worked for me…

- 1 — Be **empathic**
- 2 — Be **opinionated**
- 3 — Be **intentional**
- ***So… what opinions should I have?***

@curtiseinsmann

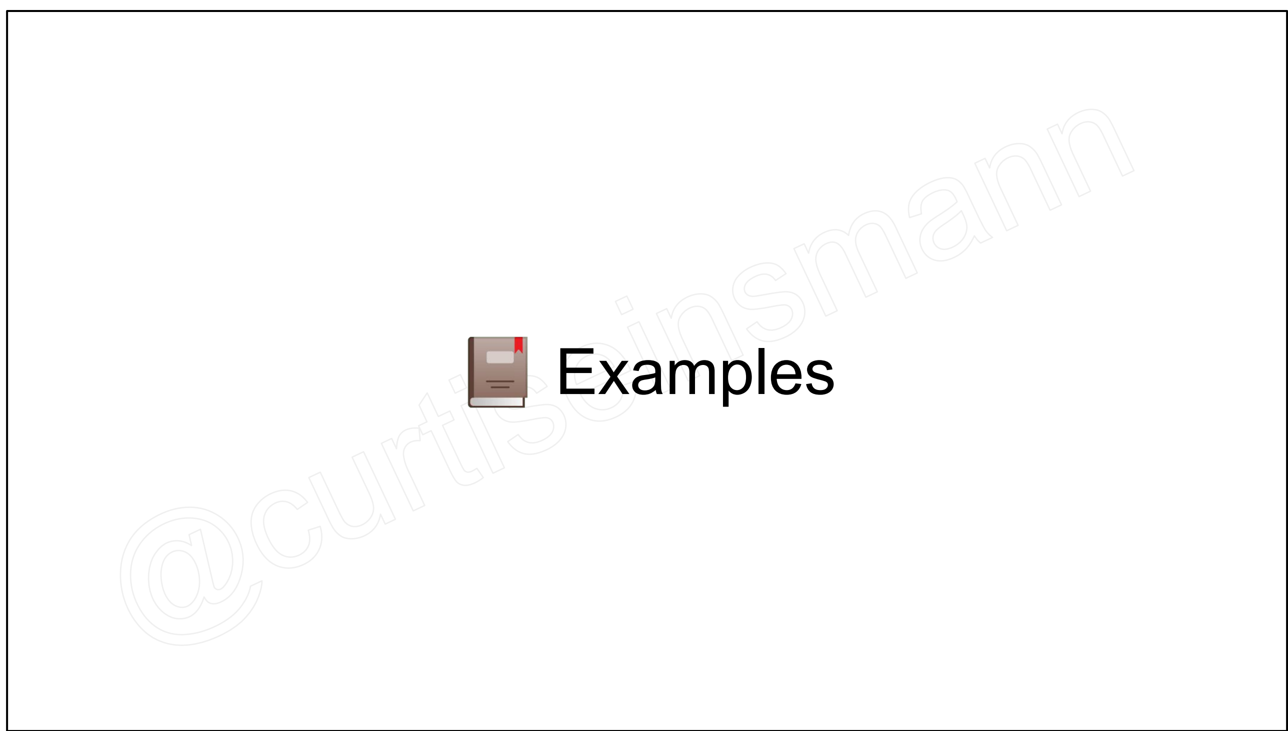
At a high level, these are the keys to writing readable code.

First you need to be empathic. You need to care. You want the readers of your code to be able to read it and maintain it for years and years. You want to make their life easier. You want to value the time of the reader more than your own.

Have opinions. Many times when you submit an assignment at university you're just trying to get it to work. Now that you want it to be readable, and readable is opinionated, you need to start having these opinions. Maybe you don't have opinions because you haven't seen a lot of code. You need to do learn about code principles. What makes code readable. Read Clean Code. Read code complete. Read blog posts and articles about what is clean code. What's readable vs. not readable, etc.

Once you have opinions, these opinions need to be reflected in your intentionality. Why did you write the code you did. Why did you make certain decisions. Every line of code you write should be intentional and true to your opinions. Your coworkers may not agree with your opinions. But that's OK, over time, you will learn what the team's general opinions are and craft code to that mindset.

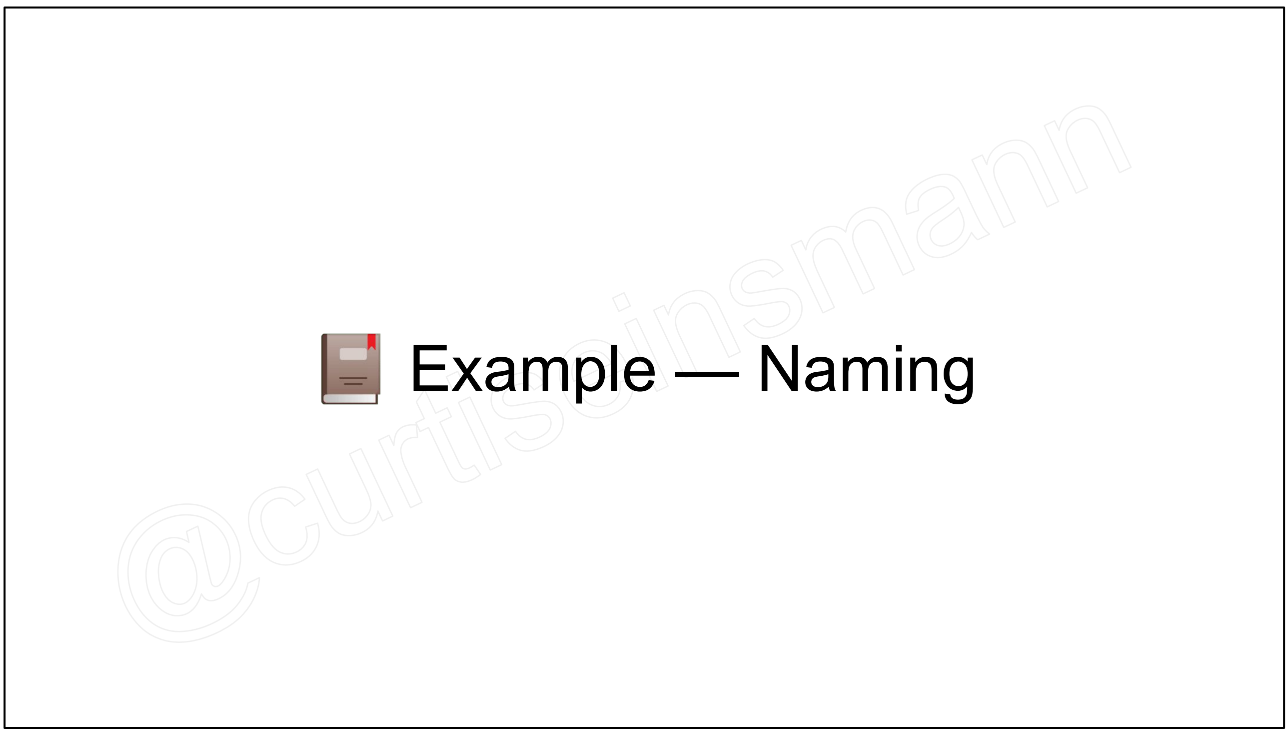
So the key to all of this is what opinions should I have? Now, as mentioned, there are entire books written on readability. A full enumeration of readability principles is outside the scope of this course. We would be here for days. [Next slide]



So what I'm going to do is show you some examples. I'm going to show you some side by side examples of code. These are examples that have been inspired from across the internet in random places. They discuss different readability principles. We're going to look at these principles in the context of the 3 pieces we discussed earlier: be empathic, be opinionated, and be intentional.

Just some housekeeping. Don't try to run the code, these are simply small samples. I will change up the language in the examples. I use JS in some, I use Java in some. Don't worry if you don't know those — these examples are language-agnostic.

[Next slide]



@curtisoinsmann

In this example we'll emphasize the principle of naming. This example was inspired by an article, and I'll link it in the slides.

Inspired by:
https://bangjelkoski.com/blog/naming-and-its-importance-in-programming/

[Show VS Code]
/naming

[show 0-numbers.js]
This example is javascript.
So here I'm showing a small function. It's called getList. Go ahead and read through the function and understand what it does.

If you read through it, you can see that it's taking an input of list, which is assumed to be a list of integers. It's using the modulo operator to check if the number is even. If so, the number is copied into the new list. Eventually the new list is returned.

So that function is pretty easy to figure out. To read through it, maybe it took you about 5 to 10 seconds. But you were able to figure it out.

[show 1-numbers.js]
Now here's the same function on the right, with everything named differently. How fast did it take you to read that one? Your eyes probably read through it rapidly. You

probably understood the code in 1 to 2 seconds, maybe less than a second.

This is the power of naming. It makes everything clear. Saving 5 seconds isn't a lot of time, but can make an incredible difference in a large codebase where you have to read thousands of lines.

So let's talk about these in terms of our keys to writing readable code.

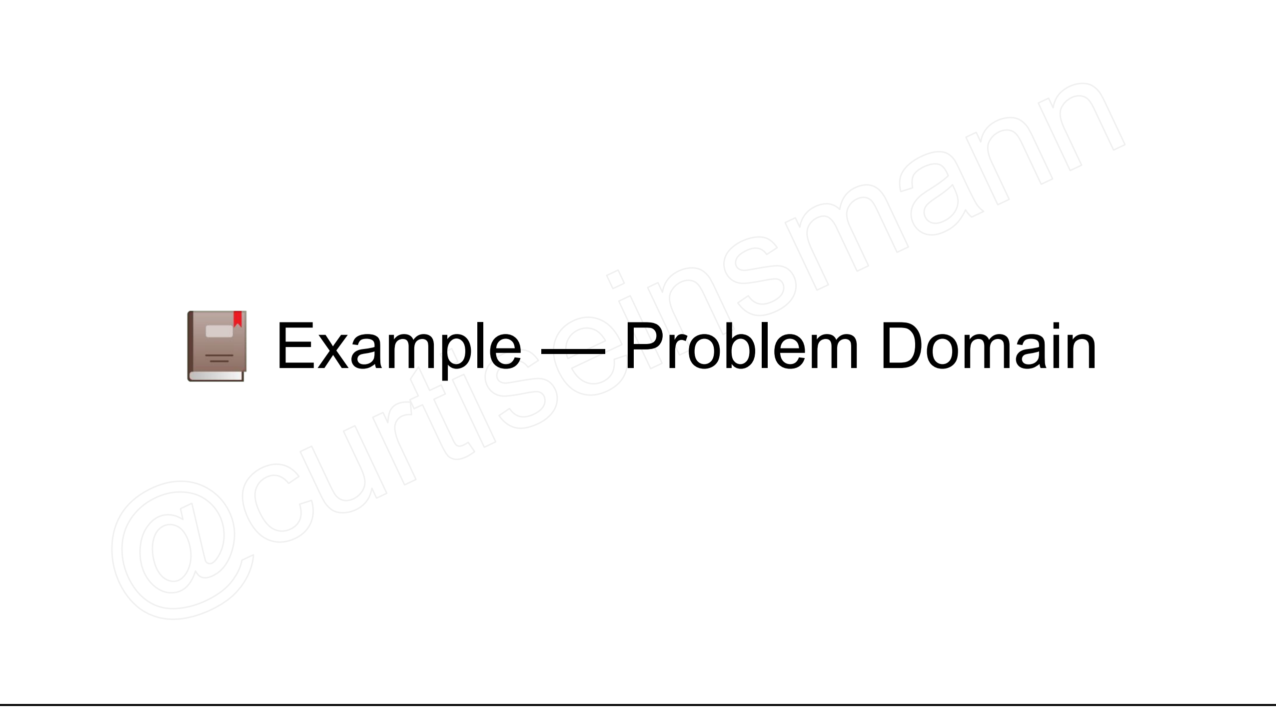
Be empathic: The author values the reader's time over their own. It may have taken them a bit longer to think of and name everything correctly. Not much — but it saves a lot of time in the future.
Be pinionated: The author has the opinion that better names lead to better readability.
Be intentional: The author reflected this belief in their code. Each name here was intentional — even the number within the loop, even though it's a temporary variable that's only used in a couple lines, was named intentionally.

[show 2-numbers.js]
And because I know somebody is going to point it out, since this is javascript, this code can be refactored into something like this, using the filter operator. And now since this function is only a couple lines long, there's a good argument around abstraction that this function likely shouldn't even exist — it should simply be used inline with whatever is calling this. But those are different opinions. I'm not making a right or wrong call here — I'm saying that choose an approach, and make an intentional choice when you write code.

```
function getList(list) {
  const list1 = [];

  for (x of list) {
        if (x % 2 === 0) {
        list1.push(x);
        }
  }

  return list1;
}


function getEvenNumbers(listOfNumbers) {
  const evenNumbers = [];

  for (number of listOfNumbers) {
        if (number % 2 === 0) {
```

```
            evenNumbers.push(number);
            }
  }

  return evenNumbers;
}
```

```
function getEvenNumbers(listOfNumbers) {
  return listOfNumbers.filter((number) => {
            return number % 2 === 0;
  });
}
```



@curtiseinsmann

In this example we'll expand on the principle of naming, but we'll specifically focus on the problem domain. It was inspired by my work at Gumroad, which is likely where you bought this course. But this is my own example code that I made up.

[Show VS Code]
See /domain

[show Domain1.java]
This example is in Java.

Here I'm showing a class called User. I've removed the constructor declaration for brevity. We can assume that calls to functions exist.

The User has a function called makePurchase, which takes in a Course as input. We store the seller of the course in a local variable. The payment is charged after getting the total price of the course. A confirmation email is sent to the User. A confirmation email is also sent to the seller.

Now at first glance there doesn't seem to be anything particularly wrong in this code. And since you're not entirely familiar with the business domain of Gumroad, you may not immediately see anything wrong here.

The problem with this code is that these entities, represented by classes and variables in the code, don't match the business domain.

User is not accurate. I'm a user of the Gumroad platform. But it wouldn't make sense for my user representation to have a makePurchase function. I'm a creator on the platform, so I sell, I don't purchase. Of course I could make purchases, but in doing so I'd be acting under a different role of my usage. The correct term here is Customer — customers are the ones who purchase things on Gumroad.

Course is not accurate. Sure you bought a course, but one can sell all kinds of different things on Gumroad. Books, templates, memberships, anything. We probably wouldn't duplicate this method for those kinds of products — the purchase will likely follow these steps no matter what they're purchasing. The correct term instead of course is Product.

Seller is also not accurate. Not all creators on Gumroad are selling things. There are many creators that give access to files or downloads that don't charge at all. Or they're using it to manage an email list and workflows. Creator is the more accurate term here.

[Show Domain2.java]

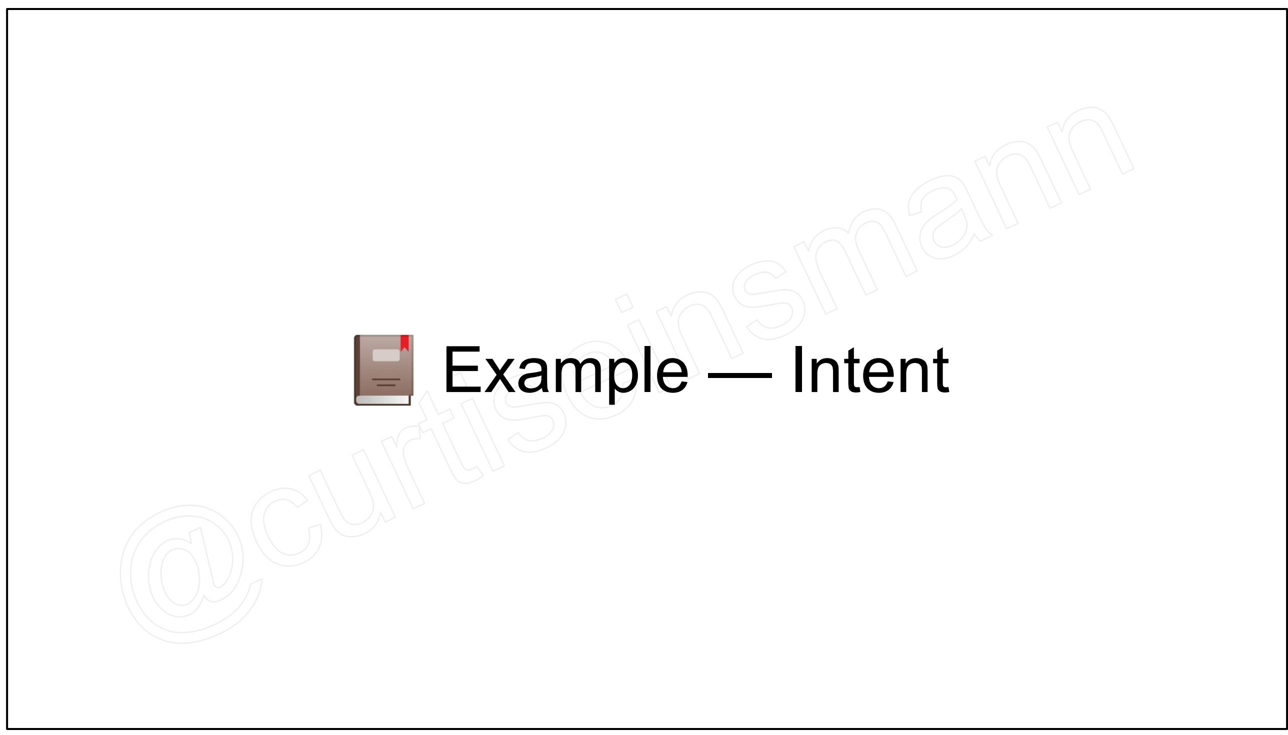
Here is the updated code with the more accurate terms. I've added some comments at the top to emphasize my points. But these comments wouldn't appear in the actual code.

Now let's talk about this in the context of our 3 keys to readability:

Be empathic: The author of this code wants the readers to understand the business problem domain.
Be opinionated: The author of this code has the opinion that terms like User, Seller, and Course are inaccurate in the context of this function.
Be intentional: The author of this code has intentionally chosen the names for this class and other entities to match the business domain.

```
public class User {

  public void makePurchase(Course course) {
          Seller seller = course.getSeller();

          chargePayment(course.getTotalPrice());

          Mailer.sendConfirmationEmailTo(
          this.email, course);
          Mailer.sendConfirmationEmailTo(
```

```
        seller.getEmail(), course);
  }
}

// Gumroad business domain:
// Creator — person who creates things on Gumroad (Curtis)
// Customer — person who buys things on Gumroad (you)
// Product — the artifact that gets bought

public class Customer {

  public void makePurchase(Product product) {
        Creator creator = product.getCreator();

        chargePayment(product.getTotalPrice());

        Mailer.sendConfirmationEmailTo(
        this.email, product);
        Mailer.sendConfirmationEmailTo(
        seller.getEmail(), product);
  }
}
```



In this example we'll take a look at the principle of intent. This example was inspired by an article, and I'll link it in the slides.

Inspired by:
https://www.freecodecamp.org/news/how-to-make-your-code-better-with-intention-revealing-function-names-6c8b5271693e/

[Show VS Code]
/intent

[show coding-tasks.js]
This example is in javascript.

This example shows a function which does some manipulation on the input for display. The goal is to display incomplete coding tasks. First it filters the tasks. It looks for tasks that are not completed and are of type coding. Then it maps each task to a javascript Object with keys fo title and userName. Then it sorts the tasks by username.

Now you might be reading through this code, and it's small, so you can understand it. But maybe the predicate in the filter portion is difficult to read. It's not immediately obvious what's being filtered here. Or maybe it's not clear why the map is being composed into JS Objects. Or maybe it's not clear in which direction the data is being sorted in the sort block.

[show 2-coding-tasks.js]
Now here on the right you can see that you can add a bit more intent around these things with the naming of functions. The author of the code on the right has split the actions up into multiple functions to highlight their intent. It’s now much more obvious that the code is filtering by incomplete coding tasks, because of the function name.

Let’s talk about these in terms of our key principles to readability:

Be empathic: The author wants the reader to understand what’s being filtered, mapped, and sorted immediately without having to reason about it.
Be opinionated: The author thinks that English function names reveal intent more than the original.
Be intentional: the author spend time breaking up this code into separate functions based on that belief.

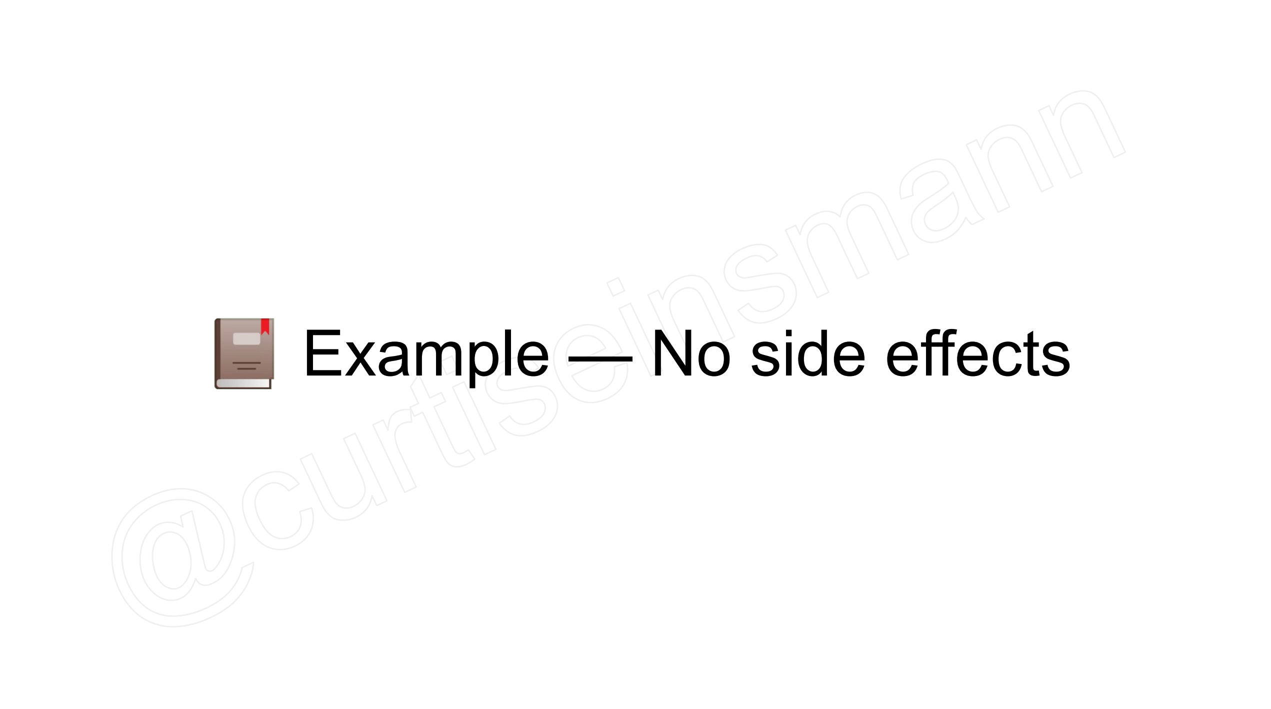
Now this example is very interesting to me. Recall that I got its inspiration from an article. Well, at this point in time of my career I would disagree with the author of the article. But 2 years ago I would’ve agreed with them

This optimization reflected by the code on the right would be in favor of Clean Code. However, after reading A Philosophy of Software Design, I’ve learned that these functions are very shallow — they’re not doing much. They’re only one liners. One liners should simply be in line with the actual code — there’s no need to break this up into functions.

So which one is best? That’s up to you. This is one of the points I was trying to highlight. This is a subjective discussion. The point is that when you’re writing code, you should have an opinion on this. And you should write the code to reflect your opinion. Don’t just write code just to write it.

```
function displayIncompleteCodingTasks(tasks){
  return tasks
        .filter(task => !task.completed &&
        task.type === "CODING")
        .map(task => ({
        title : task.title,
        userName : task.user.name
        }))
        .sort((task1, task2) =>
        task1.userName
        .localeCompare(task2.userName));
}
```

```
function displayIncompleteCodingTasks(tasks){
  return tasks.filter(isIncompleteCoding)
          .map(toViewModel)
          .sort(ascByUserName);
}
function isIncompleteCoding(task) {
  return !task.completed &&
                                task.type === "CODING";
}
function toViewModel(task) {
  return {
          title : task.title,
          userName : task.user.name
  }
}
function ascByUserName(task1, task2) {
  return task1.userName
          .localeCompare(task2.userName);
}
```



In this example we'll emphasize the principle of no side effects. This example is self created.

[Show VS Code]
/effects

[show 1-side-effect.rb]
This example is in Ruby.

Here I'm showing a small function called check_password. Given a username and password, the function checks for a match of what's stored by the application.

But if you read this function closely, checking the password is not the only thing the function does. The function increments the number of attempts if the password is wrong. This takes the action of a write to the database. By the function name alone, it's non-obvious that this will happen. A caller using this function wouldn't immediately know this is happening — they'd have to investigate.

[show 2-side-effect.rb]
Now on the right here I'm showing the same function with the updated name.

Now this comes from a principle that function names should say what they do. This is a tricky one — some function names are so complex that they can't always say

everything they do. And making the name longer can sometimes increase cognitive load. I currently have the opinion that the example on the right is better than the left. But with a longer function, it all just depends.

So let’s talk about this example in the context of our 3 keys to readability.

Be empathic: Author doens’t want readers to miss the side effect of a database write.
Be opinionated: Author thinks that making the function name explicit will clear up this confusion.
Be intentional: Author purposefully names the function to something longer.

```
def check_password(username, password)
  user = UserModel.find_by_username(username)
  return false unless user

  if user.password != password
          user.increment_attempts
          user.save!
          return false
  end
  return true
end
```

```
def check_password_and_track_attempts(username, password)
  user = UserModel.find_by_username(username)
  return false unless user

  if user.password != password
          user.increment_attempts
          user.save!
          return false
  end
  return true
end
```

📕 Example — Upholding Conventions

@curtisoinsmann

In this example we'll emphasize the principle of upholding conventions. This example is self-created.

[Show VS Code]
/conventions

[show conventions.js]
So I'm not going to show you side-by-side comparisons for this principle. Rather, I'll be comparing code within the same file.

This first example is in JS. At the top we have a function to print a string. It's using promises. When the promise is resolved, the string is printed.

These bottom two functions are doing the same thing as each other. The top is resolving the promise using the .then syntax. The bottom is resolving the promise using the await syntax.

My point here is that there are multiple ways to do this same thing. But as a reader of the code, if you saw the printString function being used this way within the same file, you'd have a temporary moment of confusion. You'd have to context switch between the syntax of the .then and the await. So when deciding between the two, pay attention to what the rest of the codebase is using. If .then is used, use that. If await is used, use that.

[show Conventions.java]

Here's another example of two functions that do the same thing. This time, in Java. The top uses streams and the bottom uses for loop. Assuming that memory is not a concern, favor what the rest of the codebase is using — or at least, whatever the code nearest to what you're modifying is using.

[show conventions.rb]

This is an example reiterating the same point in Ruby. Specifically, in Ruby rspec testing files. describe and context do the same thing here. Also note that the top uses the MyClass while the bottom usees the described_class. Prefer to follow the conventions.

Let's put these examples in the context of our keys to readability.

Be empathic: The author of code doesn't want readers to context switch between different conventions for doing the same thing.
Be opinionated: Following existing conventions reduces context switching.
Be intentional: Choose the conventions that are already present in the codebase.

```
function printString(string){
  return new Promise((resolve, reject) => {
   console.log(string)
   resolve()
  })
}


function printAll(){
  printString("A")
  .then(() => printString("B"))
  .then(() => printString("C"))
}
printAll()


async function printAll(){
  await printString("D")
  await printString("E")
  await printString("F")
}
printAll()
```

```
public class Conventions {
  public List<Person> filterEligiblePeople(
          List<Person> people) {

          return people.stream()
          .filter(p -> p.getAge() >= 21)
          .collect(Collectors.toList());
  }
}
```

```
public class Conventions {
  public List<person> filterEligiblePeople(
          List<Person> people) {

          List<Person> eligiblePeople = new ArrayList<>();
          for (Person person : people) {
          if (person.getAge() >= 21) {
          eligiblePeople.add(person);
          }
          }

          return eligiblePeople;
  }
}
```

```
describe MyClass do
  describe "when the input is nil"
          it "returns 0"
          expect(MyClass.call_function(nil)).to be (0)
          end
  end

  context "when the input is nil"
          it "returns 0"
          expect(described_class.call_function(nil)).to be (0)
          end
  end
end
```

Recap — Principles to write better code

- Objective vs. subjective
- 3 keys to write better code:
    - Be empathic
    - Be opinionated
    - Be intentional
- Further reading:
    - Clean Code
    - A Philosophy of Software Design

@curtiseinsmann

OK let's we've come to the end of this submodule. Principle to write better code. Let's recap!

We talked about objective vs. subjective. Your code needs to work first and foremost. That's the only way you're going to get your CRs approved.

But so much of CRs is around subjectivity. It's around opinions. So you need to have opinions, and be constantly evolving them.

Readability principles are outside the scope of this course. But we did go over a few. We put them in the context of these 3 keys for readability.

Be empathic. Value the readers time more than your own.
Be opinionated. Have opinions on what's readable and what isn't.
Be intentional. Every line of code that you write should have a decision behind it, backed by your opinions.

Further reading, I will link in the slides.

That's it for principles to write better code! In the next module I will walk you through a step-by-step methodical, intentional process to write better code. We'll go from task to code review. Go ahead and click into the next module and I'll see you there.

Process to write better code (step-by-step)

⚙️

@curtiseinsmann

What's up! Welcome to this submodule. Now that we know some principles to write better code, this submodule covers a process to write better code in a step-by-step way.

Main blog post: https://hashnode.com/draft/6149222c17da986cea648998

Process to write better code — why?

- Begin with the end in mind
- There’s much more to coding than writing the code
- Demystify how people code

So, why do we care about the process to writing better code?

There’s a principles from 7 habits. Begin with the end in mind. We want to get it approved in the first review. And we will work towards that goal every step of the way.

Coding is more than writing code. Completion of a task involves so many other steps. But these specific steps are rarely talked about. A solid process results in a solid code review.

Demystify how people code. Not all developers have access to a pair programming culture, and many organizations don’t operate with two people looking at code simultaneously. So the exact process of how people code can be a mystery. When I was an early career developer, I found myself tackling tasks on my own. It took a while for me to get into a rhythm with a repeatable process that works.

⚙️ Process to write better code — Take on small tasks

Bad task:

- ❌ `UpdateWidget` API

Better, but not perfect:

- ✅ Database model
- ✅ API request model
- ✅ Data Access Object (DAO) implementation
- ✅ Auth
- ✅ API implementation

@curtiseinsmann

A good code review starts before any code is written. Remember, a small PR is easier to review, and is more likely to be approved and ship quickly. But it's hard to do this if the task is huge. You want to take on small tasks.

Example of a bad task: UpdateWidget API. This is simply too large in scope. If you think about all the work it takes to create a whole API, this will inevitably be a large PR.

This breakdown is better, not perfect:
<Enumerate>

Now this isn't a perfect breakdown that'll work for all APIs. There are a lot of assumptions here. Some of these may be executed in parallel, some sequential. It depends on the system and API you're working on. But these tasks can all be separate, deployable PRs.

We are not always leading the project and breaking down tasks. Maybe it's a product manager, or tech lead. But you should do as much as you can to be engaged in the breaking down of the tasks that you're doing to work on. Over time, you should be able to manage, break down, and perform your own tasks yourself.

Sometimes your initial task breakdowns don't translate into well-scoped code reviews. And that's OK. Iterate and learn as you go.

These will also be easier to estimate if a time estimation is needed.

⚙️ Process to write better code — Do the right thing

- Descriptions tell a story — not 100% reality
- Example:
  - *"We ran into a PayPal unavailable error when charging a customer for a subscription. We attempted to retry the charge every hour. As a result, the customer kept getting subscription failure emails.* ***We should reduce the maximum number of retries to prevent these repetitive emails.****"*
  - Dove into retry logic implementation…
  - But we shouldn't even be retrying!
    - For this ***specific*** exception — YMMV.

@curtiseinsmann

Now we've broken down our tasks. But then as we're working on our task, we still need to make sure we do the right task.

Because tasks tell a story. Stories aren't 100% truth.

You'll find this situation many times in your career. Some other dev (senior or otherwise) will be diving into a problem. They see what the problem is and then suggest a solution in the task description. But when you dive into the issue, it may seem like the problem is something different from what they were describing, and then you have to solve it in the right way.

So here's an example. In Gumroad, creators can create subscription products. Customers can pay for these on a recurring basis through their PayPal. We had a problem where we kept retrying on a situation that was never going to get fixed. So the fix was to limit the number of retries. However when I dove into the problem, the actual root cause of the problem was that the original PayPal error was getting bucketed into the wrong exception type. Therefore it shouldn't have been getting retried in the first place. The appropriate fix for this specific issue was to bucket the exception in the right way. Retry limiting is something that we should do, but it wasn't the actual root cause.

This doesn't mean that the engineer is bad. They just had limited context, and the story changes when you dive deep into the actual code and see what's going on. So the moral? Do the right thing. This will come up over and over again in your career.

Don’t blindly implement what somebody else is telling you to do.

Source PR: https://github.com/gumroad/web/pull/19955

Process to write better code — Prepare environment

- Latest code
- Builds and unit tests passing
- IDE functioning correctly
- Checkout to a new branch

@curtiseinsmann

Latest code — you want to avoid merge conflicts. As you're working, the codebase will keep getting updated by other people. You don't have to update every single time somebody pushes a change, but be aware of what other people are doing so you're prepared to resolve conflicts, or coordinate with them to avoid conflicts.

Builds and tests passing — these will kill your productivity. You won't know if your code caused the tests to fail. And you'll spend a lot of time trying to figure that out unnecessarily and it'll bring you out of flow.

IDE working — you know how your IDE sometimes has false warnings, maybe an add-on didn't get installed correctly, maybe it's not using the latest version of the JDK or it doesn't match your project, maybe you're writing in React without using a React add-on, etc. Take the time to get this right. It'll improve your focus and flow so much better. Your IDE will assist you if you've made a typo. If you have so many warnings you'll just ignore them and then when you actually do mess up, it won't be very obvious. Makes it more difficult to debug.

This saves time and confusion while coding. A clean, functional environment strengthens my flow and focus. I'm more likely to produce quality work.

Checkout to a new branch — this is so you can develop on this branch without running into headaches as people are updating the main branch

⚙️ Process to write better code — Read the existing code

- Understand the current paradigm
- Leverage existing examples
- Reuse opportunities
- Refactoring opportunities
- Remember — code tells truth!

@curtiseinsmann

Understand the paradigm — if team is using describe vs context in the unit tests, you'll want to be consistent. If they're using interfaces instead of inheritance in their data access layer, you'll want to take note of that. Get an idea of the design patterns being used when you go to implement something.

Look at similar implementations for what you're trying to achieve — let's say you're trying to call the Stripe API. You can look to see how the PayPal API is implemented. Maybe it's in the proxy/ folder? Maybe it massages its parameters a certain way?

Reuse opportunities — if you're implementing the UpdateWidget API, you'll want to look at GetWidget or UpdateItem to see what can be reused.

Refactoring opportunities — see if anything is poorly written. You can use the Principles we had. With refactoring, keep in mind that some teams like it in the same code review, and some teams like it in a separate code review.

Code tells truth — remember, when you're figuring out how everything works, be cognizant of things that may lie. Documentation about how things work sometimes lies. Code comments may lie. Interfaces may lie — there could be things happening under the hood. Tasks may lie too. The code will always be the source of truth for what's happening.

Process to write better code — Coding

- To TDD or not to TDD?
- Go all out
- Make it work with ugly code
  - Implementation and unit tests
- Make it right by refactoring
  - Implementation and unit tests
- Make it fast (if necessary)

@curtiseinsmann

To TDD or not to TDD? Now there's nuance here, and it's very much an opinion based discussion. My opinion has evolved on this over the years, and it'll continue to evolve for the future. Here's my current opinion. I think TDD is great for fixing bugs. Write a test to expose the bug, then fix it. But I think you should avoid writing low level unit tests until you're sure of the design of the code's structure, classes, and abstractions. Most of the time, I like writing code and unit tests at around the same time, not in a strict order. But if your organization uses TDD, that's great, stick to what your team does, unless you really want to change it.

Go all out. Don't comment things out so you can "preserve" the existing code. Just code the entire thing.

I write lots of ugly code. I try to use decent names, but only spend a few seconds on them. They help me keep track of things as I read, iterate and refactor later. I write unit tests at around the same time. I watch the test go red first, then to green. I rarely perform manual tests during this step. Too tedious. **Example**

I refactor for readability. This is where I finalize my class and function abstractions. Every decision is intentional: Why'd I choose this name for my function? Why'd I translate this exception vs. re-throwing? Why'd I log at the WARN level and not ERROR? If anything looks weird or awkward, I search for an elegant solution. This includes the implementation and the unit tests. **Example**

I perform manual tests after refactoring. This step is necessary for some applications,

not others. I make sure all the classes and functions are connected properly. I cover basic functionality. I don’t rely on catching granular edge cases with this step — these should be covered by the unit tests.

- Scope down if necessary
- Include the “what”
- Include the “why”
- Extras if necessary
- “If I had more time, I would…”

@curtiseinsmann

Sometimes your task breakdown might not be right when you first assigned yourself tasks. So you can scope down the PRs here.

I include the *why* in the code review description.
The *why* describes the problem solved by the change.
It should implicitly answer these questions:
1. *Why* is the problem is worth solving?
1. *Why* did I choose to solve it this way?
1. *Why* did I choose the tradeoffs I did?

You like your code to be approved on the first review. But maybe you didn't have time to polish it.
So call it out in the CR description. "If I had more time, I'd..."
Preemptively call out imperfections. You're sacrificing quality so you can ship — sometimes the right choice.

Process to write better code — Opening the PR

- Review yourself in the web browser first.
- Don’t open a PR at the end of the day!
- Make sure builds are passing. Make sure no conflicts.
- Seek the relentless reviewer
- Assign to somebody familiar with the code

@curtiseinsmann

I review my own draft code review before opening it. Not in my IDE — in an internet browser. This gets me into “review mode.” The change of environment enables me to read my code with more scrutiny. I hold myself to a relentless standard. I recreate the PR if I find a flaw or readability gap. Remember there are companies that track the number of revisions after you open. There’s no point in opening the code review, realizing you made a mistake, then having to fix it.

Don’t open a PR at the end of the day. I never open a Pull Request at the end of the day. I prefer to sleep on it. This allows my subconscious to reflect on the code I've written. I almost always find a flaw or readability gap the next day. I open a higher quality PR which ships faster. People are unlikely to review PRs after working hours. The time loss for opening it in the morning is negligible. Plus: Additional quality Rightwards arrow less review cycles Rightwards arrow faster merge

Make sure all builds are passing before opening. Some teams have checks for drafts to ensure Buildkite or something like that is passing.

Seek the relentless reviewer.
There’s always that one person on every team. Nobody’s work is good enough. They nitpick everything.
Find this person. Have them review your work as much as possible. The more constructive feedback you receive, the faster you'll learn.

One reviewer, maybe two. More than that is probably overkill. In GitHub you can

assign it to somebody, more on that in effective processes later.

Don’t give yourself extra revisions. Be thorough and don’t be careless.

Recap — Process to write better code

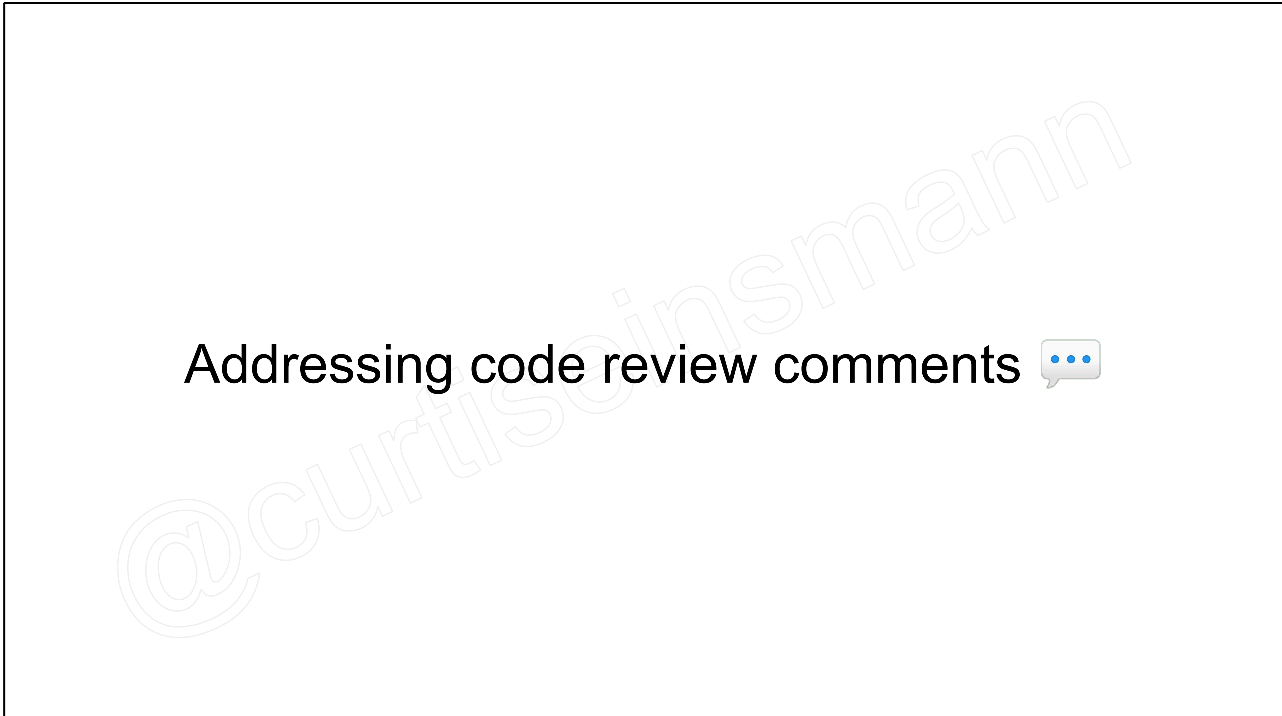
- Take on small tasks
- Do the right task
- Prepare environment
- Read the existing code
- Make it work, make it right
- Prepare and open

@curtiseinsmann

OK, let's recap!

Go through bullets…

Now in the next section, we're going to talk about how to address code review comments



What's up! welcome to this submodule. Addressing code review comments.

So you've gone through the process of taking on the task and opening the code review. Now it's time to address code review comments.

This submodule will be very heavy on the soft skills. And that's what we're looking for — remember code review is a very social process and collaborative with humans, especially this piece.

Main thread: https://twitter.com/curtiseinsmann/status/1322208044311347201?s=20

💬 Addressing code review comments — why?

- Ship faster
- Build relationships with empathic listening
- Leverage the perspective of your peers

heinsmann

Show DropBox criteria!

You want to ship code fast. You don't want to try and address somebody's comment, then you post another revision, then you have to get more comments, then you ship slower, etc.

Build relationships. You're going to be reviewing other people's code too. You want them to listen to you. It's a collaborative effort. If you listen to them, they'll listen to you. I keep referencing 7 Habits of Highly Effective people in this course, but it's a great read. So when you produce a code review, you take pride in your work. But as soon as you open it, you need to take feedback objectively. It's the team's code, not your code. This will disarm your ego and allow you to receive feedback.

My peers have unique technical paradigms. They're intelligent individuals with a variety of experiences. Their perspective exposes areas of my code to improve. Their willingness to review shows an investment in my growth.
Think about it. You have all these smart people. You need to learn from them.

And I want to piggy back of that last point. Leveraging the perspective of your peers.

I thought this picture from Visualize Value captures this so well. This is true in software development, and in life as well. You need to put your ego away. And you need to see your peers perspective as an opportunity to improve your work.

Remember we’re trying to get to truth...

💬 Addressing code review comments — getting to truth

- Read comments in monotone
- Clarify ambiguity — have a path forward
- Figure out the *reason why* behind every comment

@curtiseinsmann

First we need to understand what our teammate is saying. The truth in what they're saying.

So here's how we get to truth.

I read PR comments in monotone. I imagine they're talking with a straight face, calm tone and friendly demeanor. This nullifies my instinct to get defensive. It calibrates my reference frame. They're helping, not criticizing. (Exception: toxic people.)

Handle ambiguous comments. **Example** of what an ambiguous comment looks like. How to handle the ambiguity there.
Do **NOT** assume here. Don't even touch the code until you get an understanding of what needs to be done. You will waste a lot of time if you make assumptions. I clarify every comment. Ambiguous or half-baked comments are the enemy. I refuse to "guess" what the reviewer wants. The cost of guessing wrong is a derailment of my time. I don't touch the code lines in question until I understand the reviewer's suggestion. I clarify every comment. Ambiguous or half-baked comments are the enemy. I refuse to "guess" what the reviewer wants. The cost of guessing wrong is a derailment of my time. I don't touch the code lines in question until I understand the reviewer's suggestion.

I take time to understand the *reason why* behind every comment. This engrains the underlying principle in my brain. It helps me remember the principle in future coding situations. Even most nitpicks have a justifiable *reason why.* I take them seriously.

Take the nitpicks seriously!

💬 Addressing code review comments — acting on truth

- 3 categories
  - Correct
  - Debatable
  - Incorrect
    - *"How to I know I'm right?"*
- Pair program if necessary. *"Am I on the right track?"*
- Be thorough, as if you just opened it.

@curtiseinsmann

Now we have an idea of what our teammate is saying.

I view the feedback through an objective lens. It doesn't matter if the reviewer is a newly hired jr, or a staff engineer with 15 years of experience. Only the truth matters. The comment is either correct, incorrect or debatable.

Code review comments fall into 3 buckets: they're either correct, debatable, or incorrect.

When correct, I implement the feedback without reservations. Without reservations means you admit when they're right, and you put all your effort into fixing the mistake. Be objective and don't be afraid of admitting you did the wrong thing.

When debatable, I seek first to understand their viewpoint, then make mine understood. Know when to stand your ground, and when to give it up. I usually prefer to give it up, if the discussion is about readability. If there's an objective flaw or risk with their suggestion, I know when to stand my ground and be firm in my belief. I stand my ground when my *reason why* is stronger. If necessary I bring in a third person to resolve a disagreement.

Here's the kicker: I listen, even if the feedback is incorrect.

The thought process is: "How do I know I'm right? What caused this person to say

this? How can I clarify my code to prevent this feedback?”

This process almost always exposes a lack of clarity in my code. I leverage their perspective to improve my own work. This is a significant accelerator to the quality of my work over time.

I listen to code review comments with radically open mind. Feedback is inevitable. 1-review cycles won't always happen. I embrace comments from my peers — including the nitpicks.

I express gratitude for correct comments. It takes diligence to find flaws or ideate a more readable approach. I thank my peers when they do so. Sometimes one comment applies to many places. I fix the flaw in every applicable place within the PR.

💬 Addressing code review comments — when to talk to someone

- Remember: team level code review principles
- Abuse ❌

@curtiseinsmann

Remember team level code review principles. We talked about these in part 1 of the course. It was the first module because you should now be able to recognize if what you're dealing with is the artifact of a bad reviewer, or a bad process. And you should be able to recognize whether it's you that needs improving.

Of course there are tough reviewers who just have high standards. But then there are also abusive situations. There are times when you are simply dealing with somebody who is abusive or rude or mean or just an awful person. Be able to recognize this for what it is, and talk to somebody about it.

Recap — Addressing code review comments

- Build relationships
- Leverage perspectives
- Get to truth
- Act on the truth
- Know when to talk to someone

@curtiseinsmann

OK let’s recap! Addressing code review comments.

This course was very big on those non-technical skills. And that’s intentional — remember code reviews are a social process.

<Enumerate>

That’s it for this submodule. In the next submodule I’m going to show you some example code reviews from the author’s perspective. Specifically the ones I’ve had experience with shipping. Go ahead and click into the next submodule and I’ll see you there.

@curtiseinsmann

NOTE: Gumroad's GitHub repository is private, so GitHub links likely won't work for you.

What's up! Welcome to this submodule. In the previous submodules we talked about principles to write better code, a process to write better code, and addressing code review comments. In this submodule we're going to look at some example code reviews. These are code reviews that I personally authored at Gumroad. We'll look at some that shipped after the first review, and some that didn't. We'll talk about my thought process for producing the code. These will give you some actionable tips that you can consider when authoring your own code reviews.

https://github.com/gumroad/web/pull/20315 — example of a very simple backend API change that didn't need much context.
We'll start with a code review that was shipped in the first review. This was a simple feature addition. Gumroad has a feature which allows creators to be notified when some kind of event happens related to one of their products. For example, they can be notified of a sale, or a refund, or when a subscription starts or ends. When the specific event happens, Gumroad sends a "ping" notification to a creator's specified endpoint, with an HTTP request. The feature here was to allow creators to subscribe to membership restarts.

Let's dive into the PR itself. You'll notice that this is very concise and brief. It doesn't have screenshots or anything. For PRs, you want to provide enough context, but you don't need to overwhelm the reviewer. In this case, the code can speak for itself. It's a

backend change that doesn't need much context. But you'll notice that I'm still providing a lot of context here. I've hyperlinked the Notion and GitHub tasks for which we're using to track the work. A reviewer can read those and get context about the task. I've linked both the public docs explaining the feature, and the API docs.

I've scrolled down here. You can see that the code was approved on the first review. There were comments by the reviewers. They were all nits. Reviewers at Gumroad are very good at leaving the nits, which is encouraged, but also approving without blocking.

Now this is the view of the code. I think reading through the whole code here would be a little counterproductive, and it would probably confuse you, because you don't have context. But I do want to highlight this specific point on how I came to the solution I did. Recall that I needed to add the ability for subscription restarts. So after a little investigation I saw that I needed to add a type here. So I used my IDE to perform a search on these different types. I noticed a few different things.

- How they were stored in the database
- How they were validated in the API
- Where in the code were they set — e.g., when a subscription ends, where in the code does that actually get marked.
- Conventions around how they were tested

Remember earlier in the module we talked about reading the existing code to notice the paradigms and patterns that are used. This is an example of doing just that. And the result was a first review shipping.

https://github.com/gumroad/web/pull/19955/files — Example of a one-liner with tests that needed a lot of WHY in the context
This was another example of a code review that was approved in the first review. We're starting with a view of the code here. If you can recall in an earlier submodule, we talked about doing the right thing. We talked about how I took on a task where I discovered that we were translating a PayPal error incorrectly. This is the PR for that task.

You'll notice that this is a very small change. It's 3 lines of code and a test case. Also this looks like sensitive information, but it isn't, just some dummy test data. But yeah, this change is very small.

Now let's look at the description. And you don't have to read this. Notice how this is a very large body of text. The what is pretty brief and concise. But the Why spans over multiple paragraphs. This is because there was a divergence between the original task's description and what I actually did. All of that needed to be written out and explained. Notice how I'm hyperlinking all sorts of code lines and tasks, etc. The reviewer will read this and there won't be any ambiguity in terms of what's happening,

why the decision was made, and the background.

And scrolling down you'll see that this was very uneventful, it was approved right away. And here's my point with this PR. There are some companies — not all of them, Gumroad is not an example of one of these companies, AFAIK — but there are some companies that track the number of reviews for which you ship. Now of course this quantitative metric cannot be used without context around who the reviewers are, the complexity of the system, and other factors. But the point is, this is a metric that's tracked in some places. When you're making small one or two line changes, this is the time where you want to get your 1 review. Make sure it's tested, make sure the context is clear, and ship. Because that 1-review really drives down your average.

https://github.com/gumroad/web/pull/19646 — DB schema for fraud_refunds
[scroll down to point where it's approved] Now let's look at an example which absolutely did not get approved in the first review. This example will span across multiple pull requests. Occasionally, Gumroad has to deal with fraudulent sellers. Customers get scammed for a purchase they made, and Gumroad needs to refund those purchases. However the support team didn't have a way to distinguish the number of fraudulent refunds between other refunds, e.g., when the seller refunds for bad customer satisfaction. This was a simple feature to help Gumroad's support team track fraud refunds.

There was a discussion in the task about how we should do this. One option was to create a separate database table called fraud_refunds. One option was to add a column to an existing table called refunds. In the task discussion, we'd decided to create a separate table. So this PR you're looking at now is a migration for the creation of that table, and it's approved, and I merged it.

https://github.com/gumroad/web/pull/19665 — Fraud refund implementation
[Note: pay attention to what conversations you scroll to and show here, so as to not confuse the viewer]
[scroll down to approval] This PR is when I implemented the feature on top of that new table. We resolved a small disagreement, but then the PR was approved. And you'll notice that another engineer, Vipul, came in and gave his perspective. He was advocating for the other solution that we hadn't gone with — adding a column to the existing refunds table instead. [scroll off camera] and you'll notice here that this is the solution we're going for.

Now, I'd already implemented with the other solution. But I realized the importance of doing the right thing.

https://github.com/gumroad/web/pull/19673 — Add flags column
So I went ahead and opened this PR to add a flags column. I was able to merge that.

https://github.com/gumroad/web/pull/19684 — DB migration to drop fraud_refunds
Then I needed to open this PR to drop the fraud_refunds table that I'd previously created.

And then I made a code change to the implementation PR.

Now this example is important. Because it's true that we'd already agreed on a solution, I'd implemented it, and it was approved. And there was a reviewer that was late to the party here. But they had a strong argument, and the reviewer also agreed. Now both of these engineers, who are awesome by the way, shoutout to Chris and Vipul — they've been around the system longer and they're more experienced. So I trust their judgement. I could've complained and stuck to my guns about how I'd already implemented what we agreed upon. But that time spent is a sunk cost, and it's more important to do the right thing, especially since I'm up against a deadline.

So the takeaway from this is, sometimes you'll be in PR situations where you'll have to pivot after putting a significant time investment. But make sure you do the right thing.

https://github.com/gumroad/web/pull/20073 — Example of a change which needed a lot of screenshots, and testing steps

Now let's take a look at this PR. Gumroad allows creators to set up workflows which send emails to customers when certain things happen. Like when they purchase, subscribe, etc. I was implementing a feature that would enable creators to send an email to customers when they unsubscribe from a membership product.

This one was significantly large. As you can see there were 49 files modified here. If you have a PR this large, you may want to consider reducing scope, depending on the situation. But there were a lot of language files, images, and a majority of it was modified test code. I think there may have been around 100 implementation lines in total. So not bad.

It was a large feature. So you'll see that the why section here has a ton of context. It has testing steps. We like to drive large features like this through a QA engineer, so we need to document those. It also includes screenshots because there is some front end impact, although the change was mostly backend.

This one was not approved in the first review. You'll rarely have such a large change approved in one review. However it was approved on the second review, which is pretty good. Specifically, I want to highlight the discussions.

Now in the first review Harbaksh, who is a fantastic reviewer by the way, caught some

flaws with the code that I needed to address. I missed an edge case around variant products, which is a product which offers multiple types of deliverables.

I want to underscore this discussion. Now you don't have to read this entire thing. And you may not even understand it. I'll give you the TLDR version. Harbaksh was suggesting that I change some behavior. It was a good suggestion, but this suggestion was inconsistent with how our other workflows work, like purchases. It really could've gone either way. So after a bit of back-and-forth, I brought in a third party. That person was the head of product for Gumroad, Daniel. Shout out to Daniel, awesome product manager, former software developer at Amazon, and now solopreneur — go follow him. And Daniel gave us a path forward.

The point I'd like to make here is that I didn't keep this PR stale. I see a lot of development teams who can't come to an agreement on something. So they get lazy on driving the PR through to completion, and it gets stale. So you need to do whatever to get the PR pushed through. An in-person discussion works great. If this was an environment outside of Gumroad (which doesn't do meetings), maybe I could've jumped on a call or meeting with these people. In this case, I brought in a 3rd party and it worked out fine. This was a product decision, so I brought in the product guy. If this was an engineering decision, I probably would've brought in the head of engineering for the company.

Recap — Example code reviews

- Leverage existing code
- Get your 1-review for small changes
- Avoid sunk cost fallacy
- Drive disagreements to resolution

@curtiseinsmann

I hope you enjoyed walking through those example code reviews! We touched on a few points that I'd like to drive home here. We went through four examples. Let's talk about a main takeaway from each.

Key Performance Indicators — Write better code

- Code has fewer defects
- Shipping in less reviews
- Influencing the team's technical paradigms

@curtiseinsmann

We're coming to the end of this entire module write better code. So how do we know if we're doing well?

End of Part Three! 🐂

Congrats! You’ve reached the end of Part 3 — write better code! Time to fist pump!

Remember that MJ is the greatest basketball player ever. He was the most prolific scorer, having won the NBA scoring title 10 times. His contributions and efforts on the court were exemplary to those around him. When they saw how much effort he put into the game, his teammates followed suit.

Now that you know how to write better code, you can demonstrate the mastery of the craft to your teammates, and put them in position to win.

Part 4: Putting it all together

You’ve mastered the code review — what’s next?

@curtiseinsmann

What’s up! In this video course, you’ve Mastered the Code Review in three dimensions. You’ve learned to write better code, give better reviews, and forge a better code review process. In this final video, which I’m calling Part 4, I’m going to put it all together. I’m going to talk through how you can use what we’ve learned and apply them to other aspects in your software engineer journey. It’ll be a short one — let’s dive into it.

💬 "This isn't just about coding. This can be applied to anything."

@curtiseinsmann

As you may know I write quite a bit about software engineering on Twitter, Medium, LinkedIn. And almost weekly I get somebody saying this to me. "This isn't just about code reviews. This can be applied to anything."

And it's funny because people almost seem angry when they say this. But much of this content is very much intentional. This course was about code reviews, but it also isn't. It's a combination of soft skills, social skills, and technical skills that can be applied to many different areas, especially within software engineering.

So in the next few slides, let's talk about how those skills can be applied to other aspects.

Part 0: Master the Code Review

- Role Guidelines

@curtiseinsmann

In Part 0 remember we talked through some of the role guidelines. In each module we looked at some of the relevant guidelines and related them to the content surrounding code review.

Well keep in mind that Dropbox published these publicly, but your company probably has similar guidelines. If not, they probably have some sort of standards documented for what's expected out of you.

In 1:1 meetings with your manager, a good strategy is to walk through these guidelines. Get an idea as to which ones you're meeting, and which ones you aren't. Prioritize work so that you can fill in the gaps.

You can also consider keeping track of examples where you demonstrate good work. Keep those code reviews that were shipped fast, where you had good code review feedback. your documentation you write for code reviews. Be transparent and show these to your manager.

3 Write better code

- Principles to write better code
- Process to write better code
- Addressing code review comments

@curtiseinsmann

I'm going a bit in reverse order here. We'll start with Part 3 write better code.

We talked about principles to write better code. We talked about being empathic, opinionated and intentional. This applies to almost every artifact of work that you produce. In SWE there is always more than one right way to do something. So those 3 skills will help you make the right tradeoffs.

Process to write better code. Putting yourself in position to do highly focused work. Being thorough and seeing the work all the way through. You can't be a great software engineer without being intentionally detailed about your work, and how you go about producing it.

Addressing code review comments. Of course, this applies with any kind of feedback you receive, on anything. Your ability to listen to others will make you an overall smarter person.

Many of the things we learned in this module can go particularly well with writing and producing system design documents. High Level Design, Low Level Design. These are intensive writing tasks with a lot of tradeoffs to be made.

2 Give better reviews

- What to look for in a code review
- How to perform a code review
- Writing effective code review comments

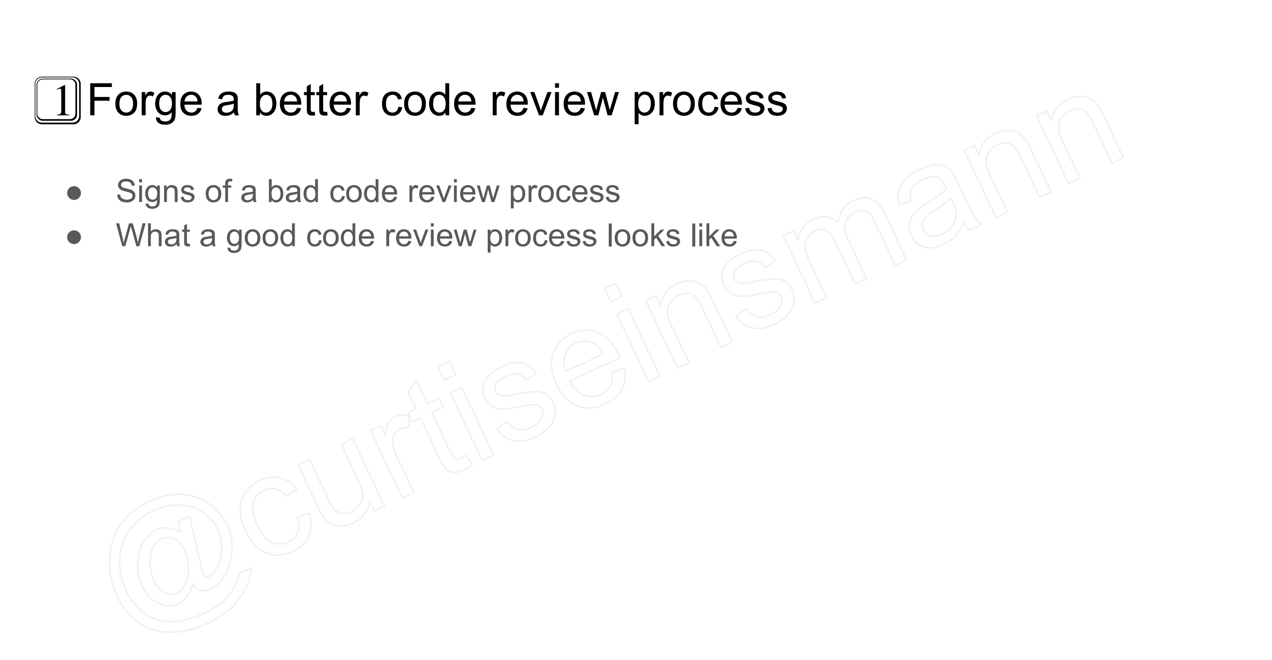
@curtiseinsmann

Part 2 was about giving better reviews.

We talked about what to look for in a code review. Remember we talked about prioritizing flaws defects. Having a high standard for readability. We talked about pragmatism vs. perfectionism. Knowing when to let things slide, and when to have a heavy hand. These are principles that come in to play when reviewing anybody else's work. In particular, design documents as well.

How to perform a code review. The main principle here was to get a basis of understanding so that you can effectively give someone feedback. Remember, you will always be able to provide insight to somebody if you understand the surrounding context and the work in depth.

Writing effective code review comments. Things like being kind, being clear, being convincing. Remote work here is here to stay in some capacity, so it'll be increasingly important to get good at this in particular. Written communication will be vitally important. Especially when mentoring junior developers.



In Part 1 we talked about forging a better code review process.

We talked through signs of a bad code review process. Most of this was about noticing inefficiencies. And instead of blaming individual skill, analyzing how the team as a whole can do better. Start noticing inefficiencies in your team's processes. Like your CI/CD pipelines. Broken or flakey tests. Onboarding processes.

We talked about what a good code review process looks like. Once you start noticing the bad, you can start fixing it with the good. Use process driven approaches to force multiply the effectiveness of your team. Introduce automation, pre-emptively prevent disagreements, proactively resolve disagreements. Remove yourself as a bottleneck, and empower your teammates.

Master the Code Review

Open in app

Download all

Videos

Slides

MASTER THE CODE REVIEW

Master the Code Review

Curtis Einsmann

Liked it? Give it a rating:

View receipt

Resend receipt

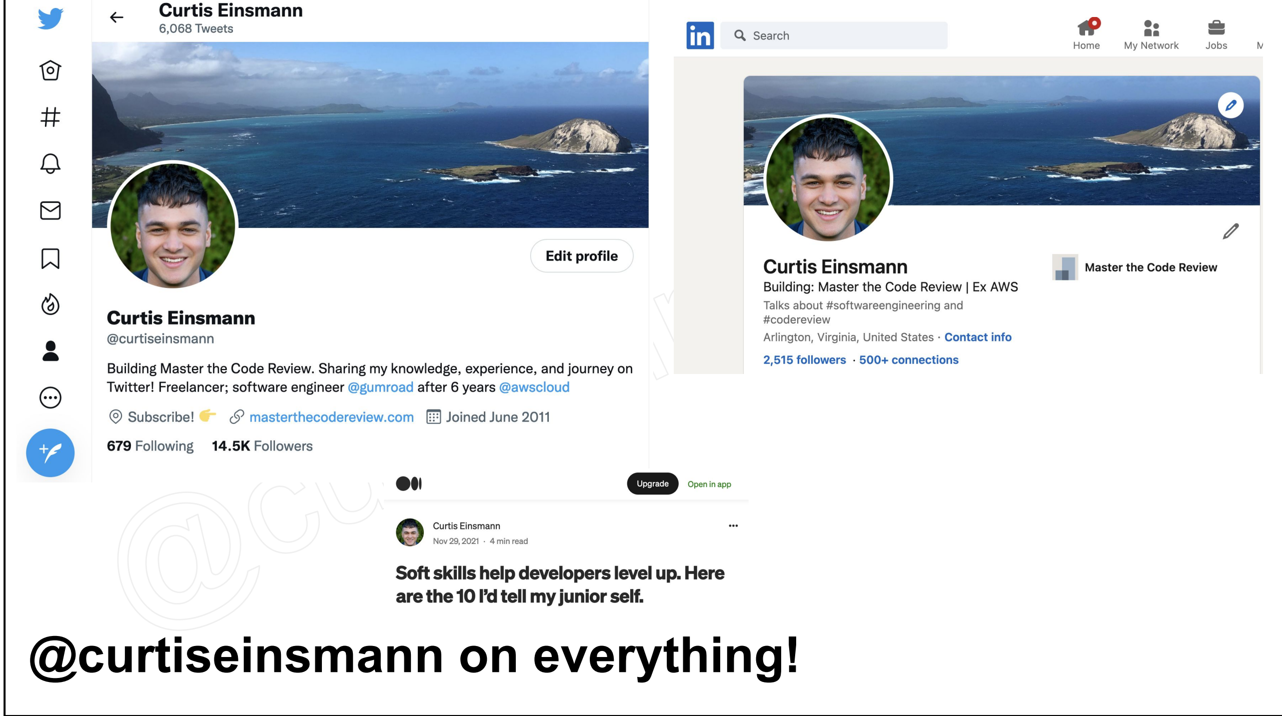
With that recap being said, I hope you enjoyed the course. I think code review is a neglected subject. Most teams are doing it wrong, and there isn't a lot of training to help developers succeed in this environment.

As mentioned, I built it for the me of the past. The one who got endless comments on code reviews, and didn't know how to review well. I hope it was useful for you!

From the download page of the course, please consider leaving a 5-star review — like I'm your Uber driver or something. It really helps.

Also if you loved the course, I encourage you to publicly share your thoughts on Twitter or LinkedIn. You can tag me, and I'll be able to take a look. It really helps to get the word out.

Also if you have any questions or feedback, simply reply to the confirmation email. Your feedback will be useful in my future projects.



A little on how to find me. This was my first product — I may do more in the future. There's a lot of beginner content out there, and the junior to mid-level SWE range is often neglected in terms of content that'll help them succeed in software development.

I know a lot of software engineering concepts and can teach you a variety of things. If you have something you'd like to learn from me, reach out and make a request or suggestion.

Think of me as your neighborhood business owner, like the bookstore, or the butcher. Except the neighborhood is on the internet. You can drop me a message or email.

It's very easy to find me, I'm @curtiseinsmann on everything. Mostly focusing on the written platforms for now like Twitter, LinkedIn, Medium. May expand to other visual platforms in the future, we'll see how this goes.

You can also reach out at curtiseinsmann@hey.com

https://twitter.com/curtiseinsmann
https://curtiseinsmann.hashnode.dev/
https://curtiseinsmann.medium.com/
https://www.linkedin.com/in/curtiseinsmann/

Additional Resources

- Google Style Guide: https://google.github.io/styleguide/
- Static Code Analysis Tools: https://www.g2.com/categories/static-code-analysis
- Code Review Tools: https://kinsta.com/blog/code-review-tools/

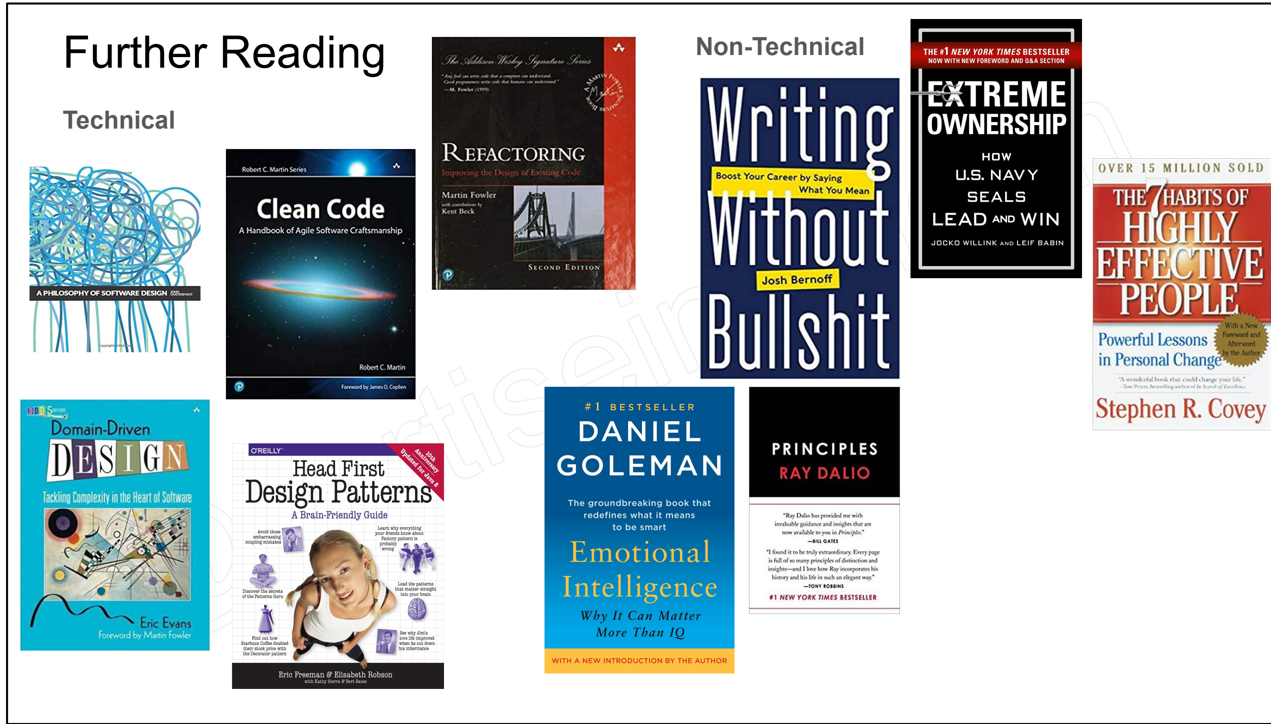
@curtiseinsmann

A couple of things I hadn't touched on in the previous modules.

If you're looking for style guides, Google has put together a great list for various programming languages. These will be very useful for your team to use.

There are many great static code analysis tools that you can use in your code review processes. I've linked a site which lists and compares them all, so you can choose from them.

Also code review tools. I mostly talked about GitHub, but there are many others. You can find the list in what I've linked.

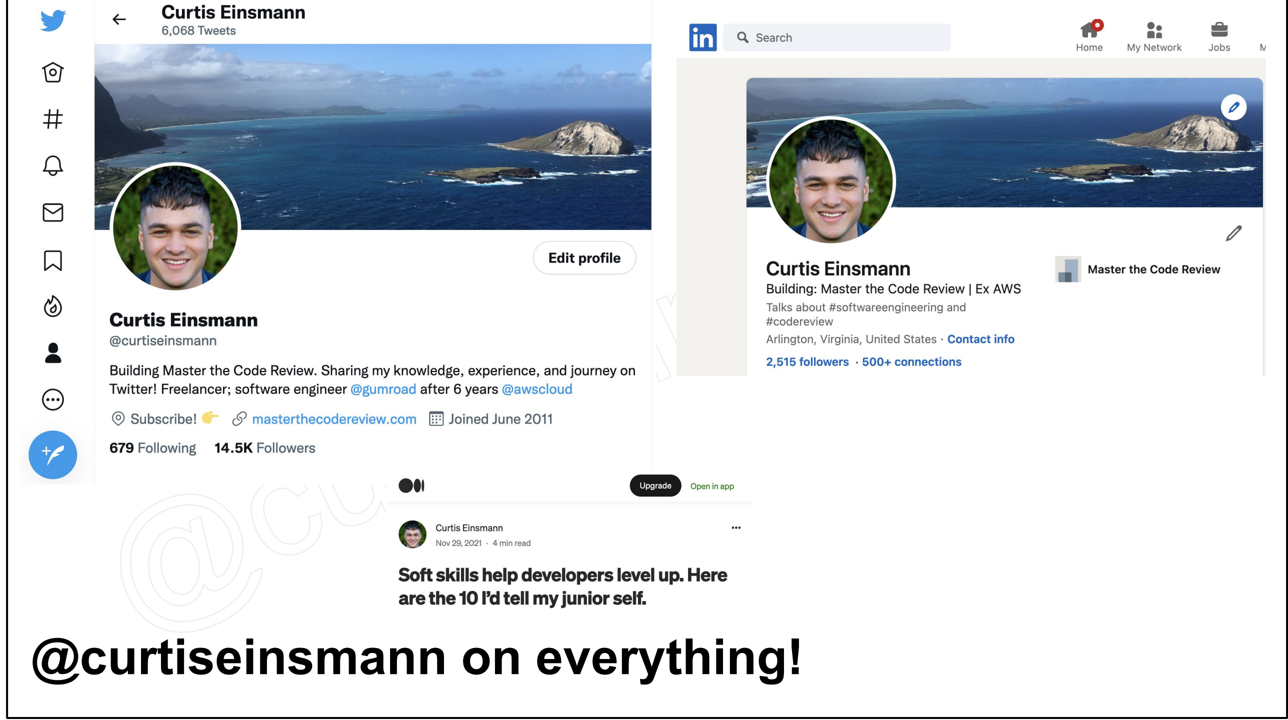


Now, here are some book recommendations. I love to read, I've probably read over 150 non-fiction books in the past 5 years. Much of the content in this course was influenced by such books. I believe technical and non-technical skills are very important. So I've given 10 recommendations here, 5 tech, 5 non-tech. They were very helpful in leveling up my SWE career, and they can be very helpful for you too.

Technical
Clean Code:
https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882
A Philosophy of Software Design:
https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201
Domain Driven Design:
https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215
Head First Design Patterns:
https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124
Refactoring:
https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672

Non-Technical
The 7 Habits of Highly Effective People:

https://www.amazon.com/Habits-Highly-Effective-People-Powerful-ebook/dp/B01069X4H0
Principles: https://www.amazon.com/Principles-Life-Work-Ray-Dalio/dp/1501124021
Writing Without Bullshit:
https://www.amazon.com/Writing-Without-Bullshit-Career-Saying/dp/0062477153
Extreme Ownership:
https://www.amazon.com/Extreme-Ownership-U-S-Navy-SEALs-ebook/dp/B0739PYQSS
Emotional Intelligence:
https://www.amazon.com/Emotional-Intelligence-Matter-More-Than/dp/055338371X



With that being said, this concludes the course. Send me an email, or message me on Twitter or LinkedIn, and I'll see you there.