Code Reviews

CSE 403 Software Engineering

Winter 2025

Today’s outline

Code Reviews

- What are they
- Why are they important
- What to consider when we do them
- Let’s practice

Course WrapUp

Final release and retrospectives next week!

Reviews (generally)

- **Review**: a constructive review of a fellow team-member's artifact (design, specification, code), providing suggested improvements
  - documentation
  - defects in program logic, efficiency
  - program structure
  - coding standards & uniformity with codebase

  … everything is fair game
- Feedback → revision / refactoring → (loop) → approval

Code reviews

- **Code review:** a constructive review of a fellow developer’s code
- A required sign-off from another team member before a developer is permitted to check in changes or new code

LGTMeow

“Let’s Go Team!”

Attribution: an excited engineer with a great attitude

“Let’s Get This Merged”

Attribution: an eager engineer with a literal translation

“Looks Good to Me”

Attribution: an engineer that probably doesn’t want to do code review, or a quick stamp of approval after a thorough code review

Why code review?

Didn’t we already test?

Code reviews are valuable

- Catch many bugs and design flaws early
- > 1 person has seen every piece of code
  - Insurance against author’s disappearance (“winning lottery”)
  - Accountability (both author and reviewers are accountable)
- Forcing function for documentation and code improvements
  - Authors must articulate their decisions
  - Authors participate in the discovery of flaws
  - Prospect of a review raises your quality threshold
- Less experienced devs get experience (without hurting code quality)
  - Pair them up with experienced developers
  - Learn by example by being a reviewer as well as a reviewee
- Promote teamwork and build trust

Let’s look at the data

- Average defect detection rates
  - Unit testing: 25%
  - Integration testing: 45%
  - **Design and code inspections: 55% and 60%** **<<<<<<<<!!**
- 11 programs developed by the same group of people
  - No reviews: average 4.5 errors per 100 LOC
  - **With reviews: average 0.82 errors per 100 LOC** **<<<<<<<<<!!**
- After AT&T introduced reviews
  - **14% increase in productivity and a 90% decrease in defects** **<<<<<!!**

(Steve McConnell’s Code Complete)

Code Reviews at Google

"All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian [now Critique] to do code reviews, and obviously, we spend **a good chunk of our time** reviewing code."

-- Amanda Camp, Software Engineer, Google

See: https://google.github.io/eng-practices/review/

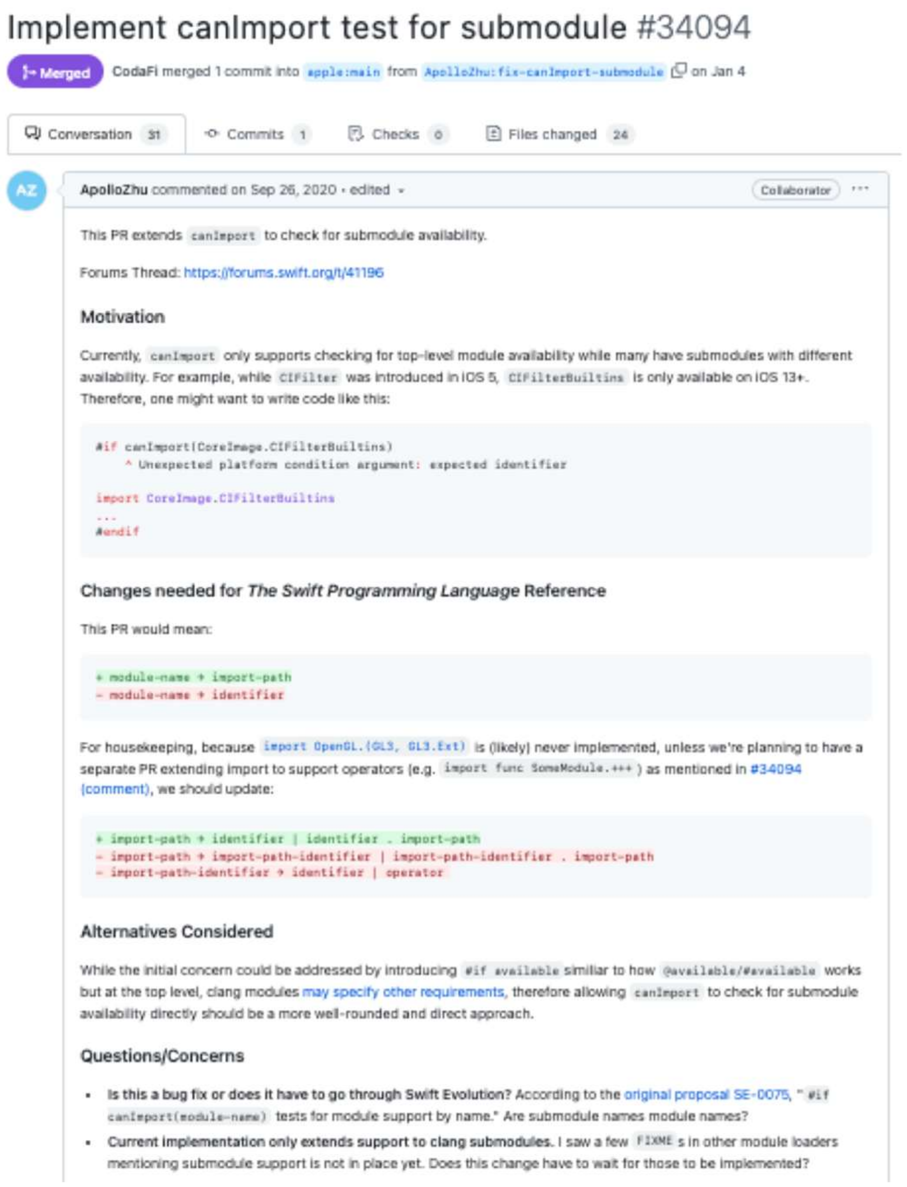
Code reviews at yelp

“At Yelp we use review-board. An engineer works on a branch and commits the code to their own branch. The reviewer then goes through the diff, adds inline comments on review board and sends them back. The **reviews are meant to be a dialogue**, so typically comment threads result from the feedback. Once the reviewer's questions and concerns are all addressed, they'll click "Ship It!" and the author will merge it with the main branch for deployment the same day.”

-- Alan Fineberg, Software Engineer, Yelp

Example dialogue

https://github.com/apple/swift/pull/34094



Implement canImport test for submodule #34094

CodaFi merged 1 commit into `apple:main` from `ApolloZhu:fix-canImport-submodule` on Jan 4

ApolloZhu commented on Sep 26, 2020 • edited

This PR extends `canImport` to check for submodule availability.

Forums Thread: https://forums.swift.org/t/41196

Motivation

Currently, `canImport` only supports checking for top-level module availability while many have submodules with different availability. For example, while `CIFilter` was introduced in iOS 5, `CIFilterBuiltins` is only available on iOS 13+. Therefore, one might want to write code like this:

```
#if canImport(CoreImage.CIFilterBuiltins)
    ^ Unexpected platform condition argument: expected identifier

import CoreImage.CIFilterBuiltins
...
#endif
```

Changes needed for *The Swift Programming Language* Reference

This PR would mean:

```
+ module-name → import-path
- module-name → identifier
```

For housekeeping, because `import OpenGL.(GL3, GL3.Ext)` is (likely) never implemented, unless we're planning to have a separate PR extending import to support operators (e.g. `import func SomeModule.+++`) as mentioned in #34094 (comment), we should update:

```
+ import-path → identifier | identifier . import-path
- import-path → import-path-identifier | import-path-identifier . import-path
- import-path-identifier → identifier | operator
```

Alternatives Considered

While the initial concern could be addressed by introducing `#if available` similar to how `@available/#available` works but at the top level, clang modules may specify other requirements, therefore allowing `canImport` to check for submodule availability directly should be a more well-rounded and direct approach.

Questions/Concerns

- **Is this a bug fix or does it have to go through Swift Evolution?** According to the original proposal SE-0075, " `#if canImport(module-name)` tests for module support by name." Are submodule names module names?
- **Current implementation only extends support to clang submodules.** I saw a few `FIXME` s in other module loaders mentioning submodule support is not in place yet. Does this change have to wait for those to be implemented?

Code reviews at

"At Wizards we use Perforce for SCM. I work with stuff that manages rules and content, so we try to commit changes at the granularity of one bug at a time or one card at a time. Our team is small enough that you can designate one other person on team as a code reviewer. Usually you look at code sometime that week, but it depends on priority. It's **impossible to write sufficient test harnesses** for the bulk of our game code, so **code reviews are absolutely critical."**

-- Jake Englund, Software Engineer, MtGO

Code reviews at

"Once an engineer has prepared a change, she submits it to this [code review] tool, which will notify the person or people she has asked to review the change, along with others that may be interested in the change – such as people who have worked on a function that got changed.

At this point, the reviewers can make comments, ask questions, request changes, or accept the changes. If changes are requested, the submitter must submit a new version of the change to be reviewed. All versions submitted are retained, so reviewers can compare the change to the original, or just changes from the last version they reviewed. Once a change has been submitted, the engineer can merge her change into the main source tree for deployment to the site during the next weekly push, or earlier if the change warrants quicker release."

- Ryan McElroy, Software Engineer, Facebook

What to consider with a code review
Best practices

First, agree on a plan with your team

- What has to be reviewed:
  - A document (requirements, specification, guide, …)
  - A coherent module (sometimes called an “inspection”)
  - A single checkin or code commit (incremental review)
- Who participates:
  - One other developer
  - A group of developers
- Where:
  - Email/electronic
  - In-person meeting
    - Best to prepare beforehand: artifact is distributed in advance
    - Preparation usually identifies more defects than the meeting
- When:
  - What’s the expected turn around time

Common for big picture reviews: holistic

- Review the **goal**, the **architecture**, the **design**, or an **entire component**
- Each reviewer focuses where their area of expertise
  - Mark up with comments
  - Identify most important issues
- At meeting, may go around the table raising one issue
  - Discuss the reasons for the current approach and possible improvements
- Author addresses all issues in comments
  - Not just those raised in the meeting

Common for code reviews: incremental

- **Each change** is reviewed *before* it is committed
- No change is accepted without signoff by a “committer”
  - Part of the “plan” is, who can do the actual commit
  - Committer is assumed to know the whole codebase well; most often is the developer of the change
- Code review can (d)evolve into a design discussion

Next, establish the goal of the code review

- Verification: are we building the system right?
- Validation: are we building the right system?

- Presence of good properties?
- Absence of bad properties?

- Identifying errors?
- Confidence in the absence of errors?

- Robust? Safe? Secure? Available? Reliable?
- Understandable? Modifiable?
- Cost-effective?
- Usable?

And clarify what exactly is being asked

A Google guideline:

“Make sure to **review every line of code** you’ve been asked to review, look at the context, make sure you’re **improving code health**, and **compliment developers** on good things that they do.”

Leverage review checklists

See: https://google.github.io/eng-practices/review/reviewer/looking-for.html

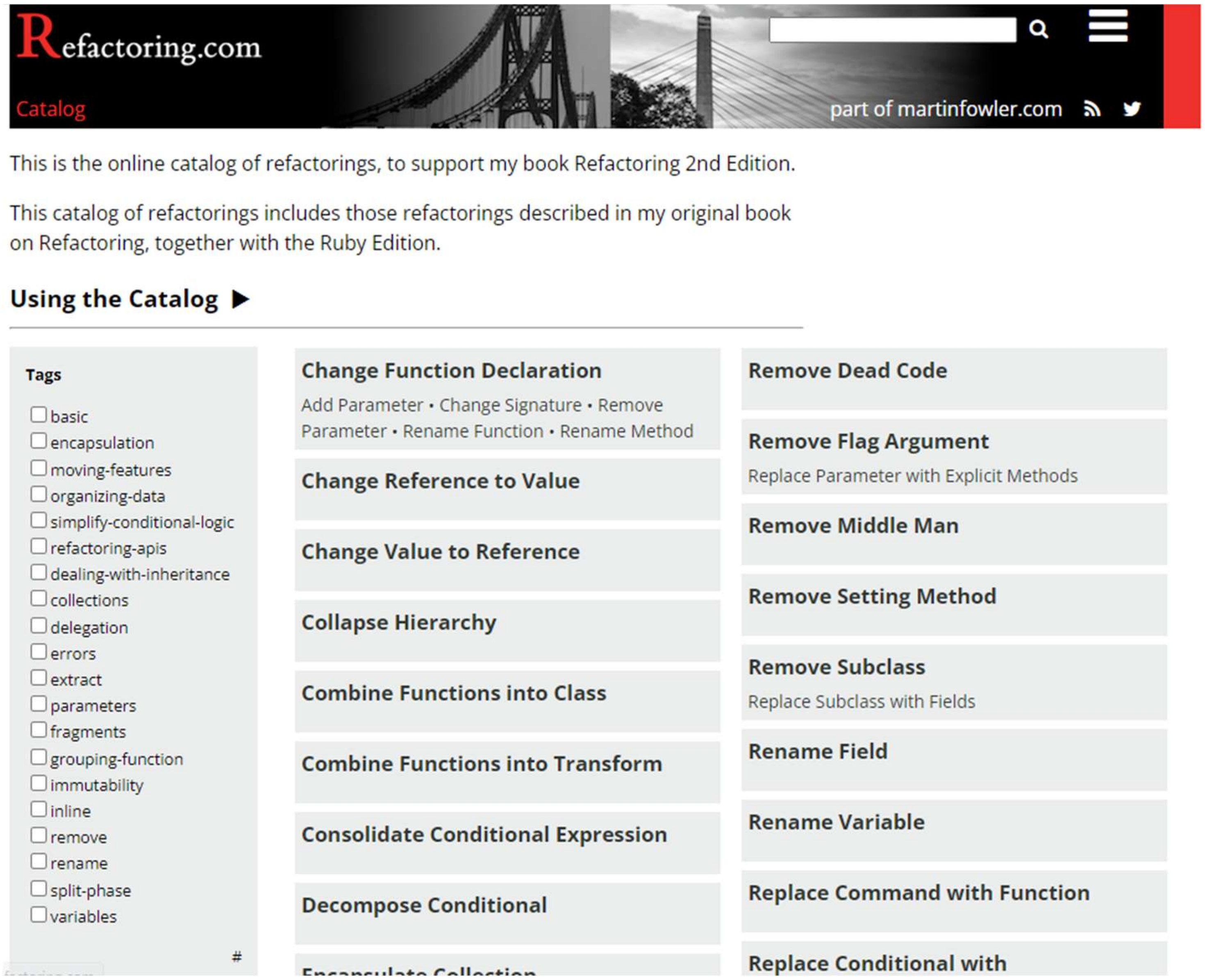
Consider if -

- The code is well-designed.
- The functionality is good for the users of the code.
- Any UI changes are sensible and look good.
- Any parallel programming is done safely.
- The code isn't more complex than it needs to be.
- The developer isn't implementing things they *might* need in the future but don't know they need now.
- Code has appropriate unit tests.
- Tests are well-designed.
- The developer used clear names for everything.
- Comments are clear and useful, and mostly explain *why* instead of *what*.
- Code is appropriately documented (generally in g3doc).
- The code conforms to your style guides.

Practice
Let’s do a code review

```
public class Account {
   double principal,rate;     int daysActive,accountType;

   public static final int STANDARD=0, BUDGET=1,
   PREMIUM=2, PREMIUM_PLUS=3;
}

public static double calculateFee(Account[] accounts)
{
   double totalFee = 0.0;
   Account account;
   for (int i=0;i<accounts.length;i++) {
      account=accounts[i];
      if ( account.accountType == Account.PREMIUM ||
           account.accountType == Account.PREMIUM_PLUS )
         totalFee += .0125 * (       // 1.25% broker's fee
            account.principal * Math.pow(account.rate,
            (account.daysActive/365.25))
            - account.principal);   // interest
   }
   return totalFee;
}
```

Good resource to learn some code styles (patterns)



Refactoring.com

Catalog

part of martinfowler.com

This is the online catalog of refactorings, to support my book Refactoring 2nd Edition.

This catalog of refactorings includes those refactorings described in my original book on Refactoring, together with the Ruby Edition.

Using the Catalog ▶

Tags

- ☐ basic
- ☐ encapsulation
- ☐ moving-features
- ☐ organizing-data
- ☐ simplify-conditional-logic
- ☐ refactoring-apis
- ☐ dealing-with-inheritance
- ☐ collections
- ☐ delegation
- ☐ errors
- ☐ extract
- ☐ parameters
- ☐ fragments
- ☐ grouping-function
- ☐ immutability
- ☐ inline
- ☐ remove
- ☐ rename
- ☐ split-phase
- ☐ variables

#

Change Function Declaration
Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method

Change Reference to Value

Change Value to Reference

Collapse Hierarchy

Combine Functions into Class

Combine Functions into Transform

Consolidate Conditional Expression

Decompose Conditional

Remove Dead Code

Remove Flag Argument
Replace Parameter with Explicit Methods

Remove Middle Man

Remove Setting Method

Remove Subclass
Replace Subclass with Fields

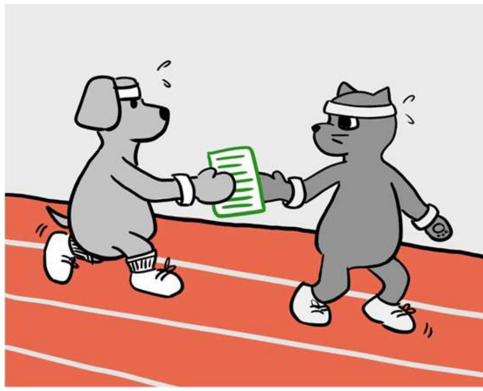
Rename Field

Rename Variable

Replace Command with Function

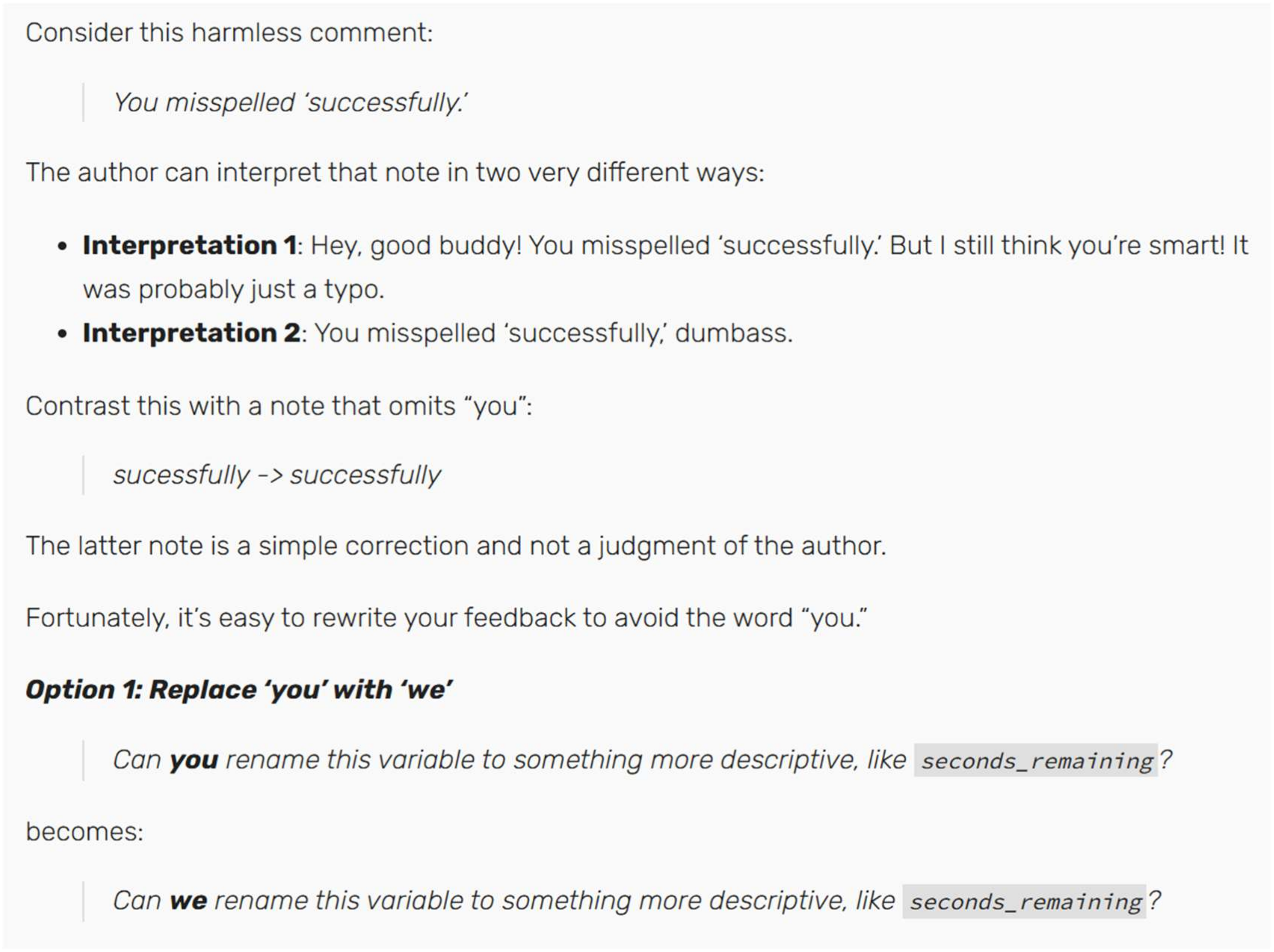
Replace Conditional with

Be a human as you do your review



1. Settle style arguments with a style guide
2. Let computers do the boring parts: linters/formatters/CI
3. Give code examples (build trust)
4. Never say “you” (focus on the code, not the coder!); “we” = team ownership
5. Requests and questions, not commands and criticism … frame it as an in-person conversation
6. Offer sincere praise
7. Incremental improvements instead of perfection
8. Handle stalemates proactively

See: https://mtlynch.io/human-code-reviews-1/



Consider this harmless comment:

> *You misspelled 'successfully.'*

The author can interpret that note in two very different ways:

- **Interpretation 1**: Hey, good buddy! You misspelled 'successfully.' But I still think you're smart! It was probably just a typo.
- **Interpretation 2**: You misspelled 'successfully,' dumbass.

Contrast this with a note that omits "you":

> *sucessfully -> successfully*

The latter note is a simple correction and not a judgment of the author.

Fortunately, it's easy to rewrite your feedback to avoid the word "you."

Option 1: Replace 'you' with 'we'

> *Can **you** rename this variable to something more descriptive, like `seconds_remaining`?*

becomes:

> *Can **we** rename this variable to something more descriptive, like `seconds_remaining`?*

```
public class Account {
   double principal,rate;     int daysActive,accountType;

   public static final int STANDARD=0, BUDGET=1,
   PREMIUM=2, PREMIUM_PLUS=3;
}

public static double calculateFee(Account[] accounts)
{
   double totalFee = 0.0;
   Account account;
   for (int i=0;i<accounts.length;i++) {
      account=accounts[i];
      if ( account.accountType == Account.PREMIUM ||
           account.accountType == Account.PREMIUM_PLUS )
         totalFee += .0125 * (       // 1.25% broker's fee
            account.principal * Math.pow(account.rate,
            (account.daysActive/365.25))
            - account.principal);   // interest
   }
   return totalFee;
}
```

Improved code (page 1)

```
/** An individual account.  Also see CorporateAccount. */
public class Account {
   private double principal;
   /** The yearly, compounded rate (at 365.25 days per year). */
   private double rate;
   /** Days since last interest payout. */
   private int daysActive;
   private Type type;

   /** The varieties of account our bank offers. */
   public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}

   /** Compute interest. **/
   public double interest() {
      double years = daysActive / 365.25;
      double compoundInterest = principal * Math.pow(rate, years);
      return compoundInterest - principal;
   }

   /** Return true if this is a premium account. **/
   public boolean isPremium() {
      return accountType == Type.PREMIUM ||
             accountType == Type.PREMIUM_PLUS;
   }
```

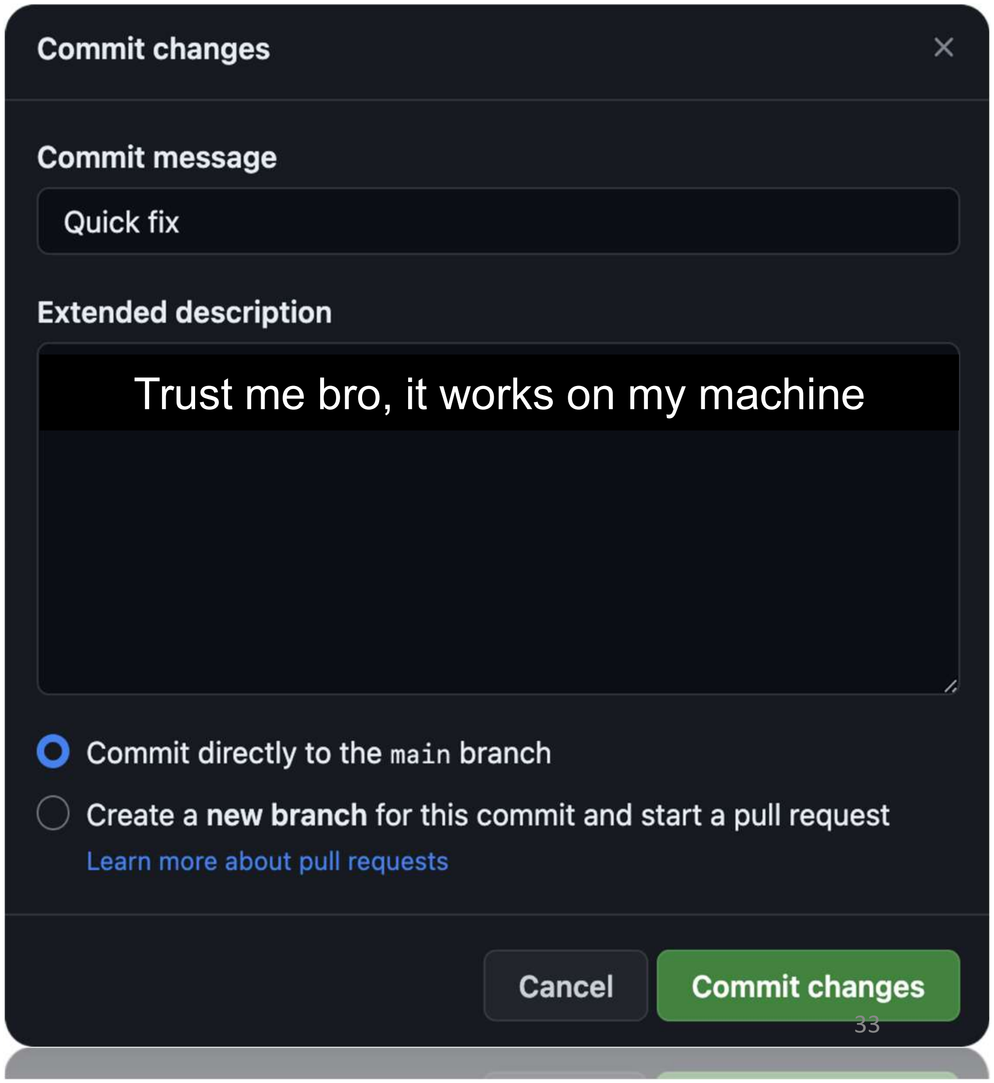
Improved code (page 2)

```
    /** The portion of the interest that goes to the broker. **/
    public static final double BROKER_FEE_PERCENT = 0.0125;

    /** Return the sum of the broker fees for all the given
    accounts. **/
    public static double calculateFee(Account[] accounts) {
       double totalFee = 0.0;
       for (Account account : accounts) {
          if (account.isPremium()) {
             totalFee += BROKER_FEE_PERCENT * account.interest();
          }
       }
       return totalFee;
    }

}
```

Writing a good pull request is also important

“What could go wrong?”

Famous last words


Commit changes

Commit message

Quick fix

Extended description

Trust me bro, it works on my machine

[✓ Commit directly to the `main` branch]

Create a **new branch** for this commit and start a pull request

Writing a good pull request

Think like a reviewer

- Use descriptive but concise title and summary
- Describe context, rationale, and alternatives considered
- Link to relevant resources (specs, issues/bug tracker, previous PR)
- Provide screenshots/recordings for UI changes

See:
How to write the perfect pull request
https://github.blog/2015-01-21-how-to-write-the-perfect-pull-request/

Writing good CL descriptions
https://google.github.io/eng-practices/review/developer/cl-descriptions.html

Congrats, 403 – it’s a wrap! (almost)

Final release demos next week

Most time should be demo’ing your live product, with a few minutes for your reflections – what you learned, what you might do differently for v2.0

Is this all now so familiar? You’ve lived it!

A SDLC defines how to produce software through a series of stages

Common stages

- Requirements
- Design
- Implementation
- Testing
- Release
- Maintenance

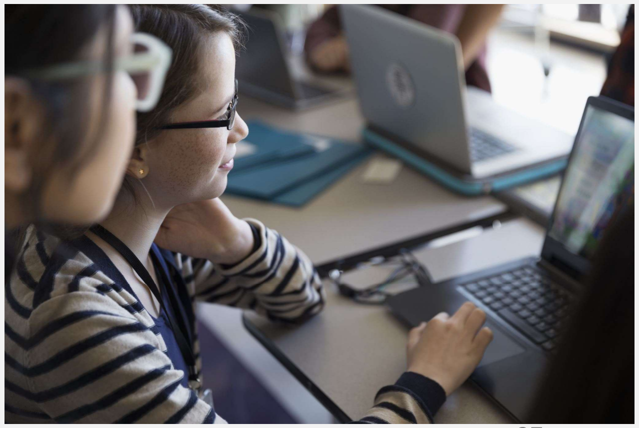
Goals of each stage

- Define a clear set of actions to perform
- Produce tangible (trackable) items
- Allow for work revision
- Plan actions to perform in the next stage

Key question: how to combine the stages and in what order

By the end of the quarter, you'll have…

- Been exposed to some of the best software development practices in use today
- Understand how software is produced – from conception to continuous delivery and release
- Developed skills to effectively collaborate with others towards a common delivery goal
- Experienced the responsibilities, issues and tradeoffs involved in making decisions as software engineers



It’s a journey that will continue

- Compare your skills today to a quarter ago
  - Bottom line: Your project would be easy for you
    - This is a measure of how much you have learned
- Your next project can be much more ambitious
- You will continue to learn
  - Building interesting software systems is never easy
    - Like all worthwhile endeavors
  - Practice is a good teacher
    - Requires thoughtful introspection
    - Don’t learn *only* by trial and error ☺

Onward!

Go forth and ship amazing products that delight your customers, again and again

- Building products and services is fun and rewarding
  - Especially when you build them successfully
- Pay attention to what matters
  - Use the techniques and tools of CSE 403 effectively

Course evaluation

- Please complete the course evaluation form online
  - Useful to future students
  - Useful to course staff
  - Useful to the department