COMS W4156 Advanced Software Engineering (ASE)

October 12, 2021

shared google doc for discussion during class

Writing Code the Right Way

We started the semester with

Software Engineering != Programming

- *Time and Change* affect the sustainability of software, how code adapts over the length of its life
- *Scale and Growth* affect the viability of software practices, how an organization adapts as it grows and evolves

But the *cost of errors* may possibly be the most significant difference between software engineering and programming

As a student, the cost of bad code is usually (at worst) a bad grade

As a professional software engineer, the cost of bad code could be millions of dollars, damage to property, and/or injuring or killing humans

Preventing Bad Code from being Released in Production

Bad code may not matter too much if it never leaves the developer's machine

Bad code committed to shared repositories will increase time and effort for *other* software engineers, but is unlikely to be humongously expensive if it never makes it to production (users)

Testing is usually the last barrier to preventing bad code from leaving the developer's machine or, if it gets to shared repositories, preventing bad code from being released into production

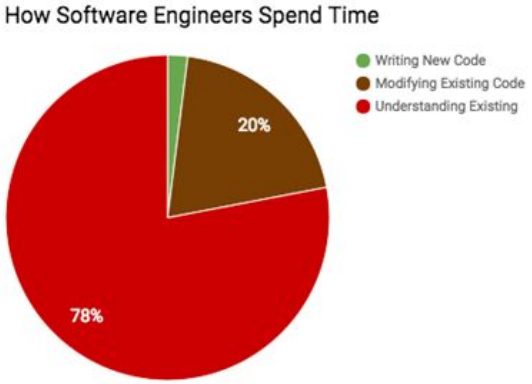
A lot of this course is about testing, but what can we do *before* testing? Or in addition to testing, since testing cannot avoid false negatives

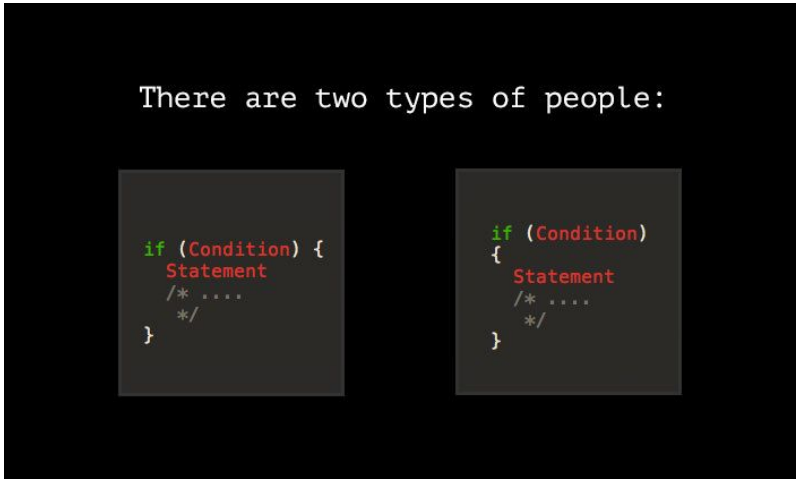
Refresher: Coding Style

Governs how and when to use comments and whitespace (indentation, blank lines), proper naming of variables and functions (e.g., camelCase, snake_case), code grouping and organization, file/folder structure

Compliance with a team's coding style helps other developers understand your code and vice versa, even helps your future self understand your past self's code

See Google's [many style guides]



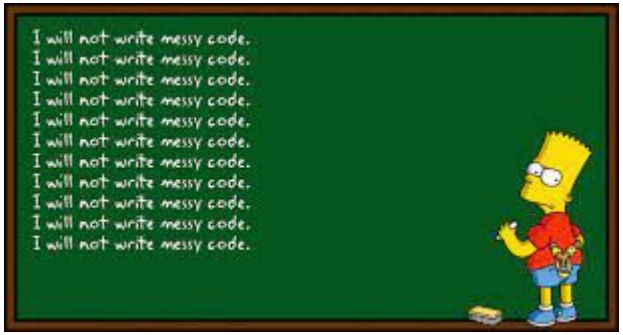


Examples

```
public class TestBadCode {

    public int calculateFoo(int x, int y, boolean increment){
        if(increment)
            x++;
            x *=2;
        x += y;
        return x;
    }

    @Test
    public void test() {
        assertEquals(calculateFoo(3, 5, true), 13);
        assertEquals(calculateFoo(3, 5, false), 8);
    }

}
```

```
switch(value) {
    case 1:
        doSomething();

    case 2:
        doSomethingElse();
        break;

    default:
        doDefaultThing();
}
```

Refresher: Style Checkers



Most languages have automated tools that detect deviations from a prescribed coding style, e.g., checkstyle

The auto-merge feature of git and other version control systems depend on style compliance to avoid superficial, format-related merge conflicts - so should always run style checking and fix any problems before commiting code changes

Running the style checker (and doing various other things) prior to each commit can be automated for github with a git pre-commit hook

What you should do

Each team should pick a coding style that has a corresponding style checker

Use the coding style for all the code in your team project

Run the style checker *before* commiting code to your shared repository

Also run the style checker on your shared repository regularly, just in case - and also because Assignment T3: First Iteration requires that you include style checker reports in your repository

Refresher: Code Documentation

See Google's Documentation Best Practices and Engineering Practices
- "*Documentation is the story of your code*"

A non-trivial method (function, procedure, subroutine, etc.) has *inline* comments that explain "why" the code is written this way or "why" it works this way - intended audience is future developers who modify

Methods have a header (Javadoc, docstring, etc.) that explains "what" the method does and "how" to use it - intended audience is future developers who use or modify

Every behavior documented in a method header has corresponding test cases to verify that behavior, and the test cases themselves are documented with what arguments the method takes, what it returns, any "gotchas" or restrictions, and what exceptions it can throw or errors it can return

Refresher: Code Documentation

Classes (modules, files, etc.) document the class as a whole, overview of what the class does and examples of how it might be used (both simple and advanced, if applicable) - intended audience is future developers who use or modify

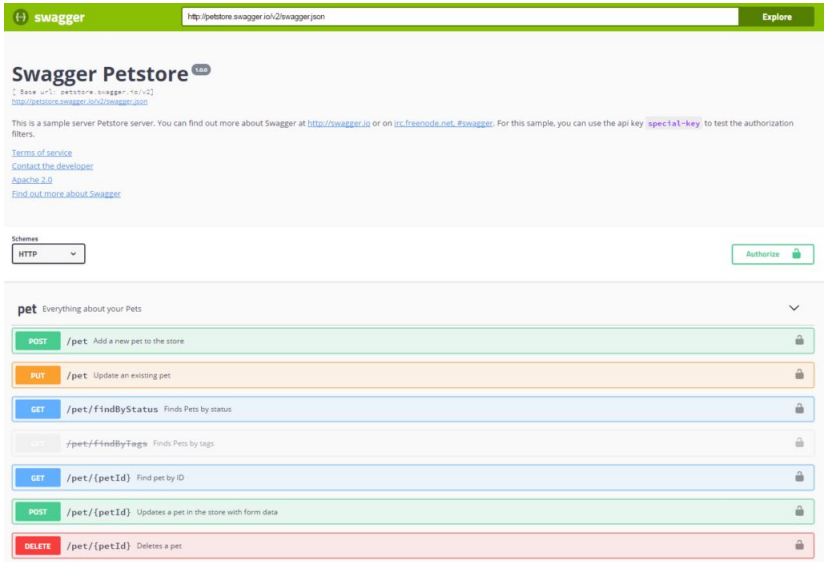
Folders (directories) have a README.md file that states What is this directory intended to hold? Which files should the developer look at first? Are some files an API? Who maintains this directory and where I can learn more?

API Documentation

OpenAPI “standard” for RESTful APIs

Javadoc “standard” for Java

Linux man-pages project



Some style checkers check that certain code documentation exists and conforms to format, e.g., checkstyle for Javadocs

There are even “prose-linters” like Vale that check some documentation content - different from spelling/grammar checkers, because they know they are checking code documentation (and can deal with code syntax in the midst of prose)

What you should do

Each team should also agree on an approach to code documentation and use it

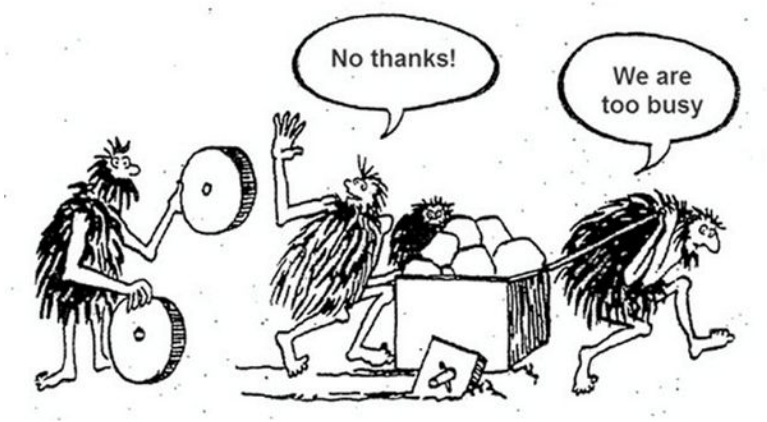
“No documentation” is not an acceptable approach!

Your repository’s top-level README.md should document your API - required for Assignment T3: First Iteration

You are not required to follow any particular format or do anything else fancy

“Technical Debt”

Cost of additional rework later on caused by choosing a fast and easy solution (bad code) now instead of a better solution (good code) that would take longer - not a big concern for a one-semester course project,
but indeed a big concern for industry software



Analogous to monetary debt - It’s ok to borrow against the future, as long as you understand that you need to pay it off

If technical debt is not repaid quickly, accumulates “interest” that adds up substantially over time - Justifies the time/effort spent on refactoring

Where does Technical Debt come from?

From developers …

A common example of incurring technical debt is when a code segment is copy/pasted from one location in the codebase to another

Copying existing code looks like a quick win—why write something anew when it already exists?

“Code clones” = identical or nearly identical code that exists in more than one location in the codebase, typically arising via copy/paste

Prioritizing Technical Debt Repayment

Eventually, those two parts of the codebase might be maintained by different developers (or even different teams)

A developer finds a bug in one of those copies and another developer finds the same or different bug in another copy

Is it better for each developer to fix each copy independently?

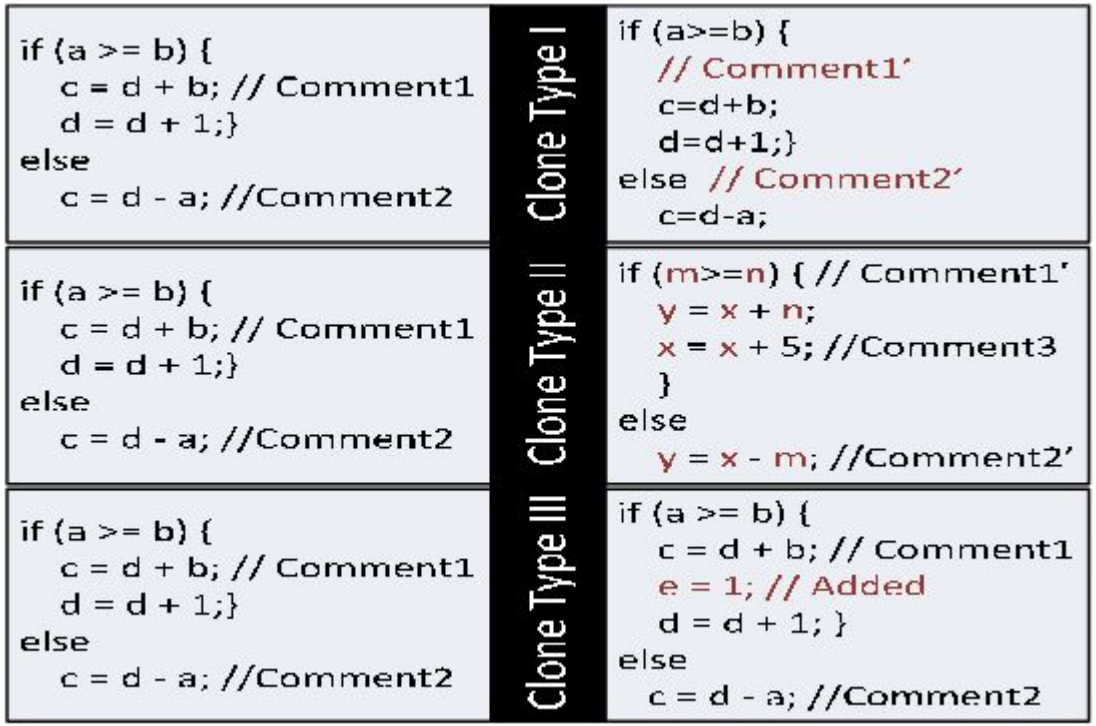
Or is it better to refactor so this piece of code appears in only one place and is called in the two original places, and then fix bugs?

Example

Code Clone Detectors

But how do the two developers know that there are other copies?

Even if it's the same developer working with both copies, the developer may not recognize that they are copies due to reformatting, identifier renaming, other divergences that accumulate over time



Code Clone Detectors

There are many automated code clone detectors that can help detect these locations

Some code clone detectors are language-specific, some work across languages, e.g., CPD

Some refactoring tools, e.g., JDeodorant, not only automatically detect a group of code clones but also help developers replace each instance of the clone with a call to a single code unit (look at beginning of video)

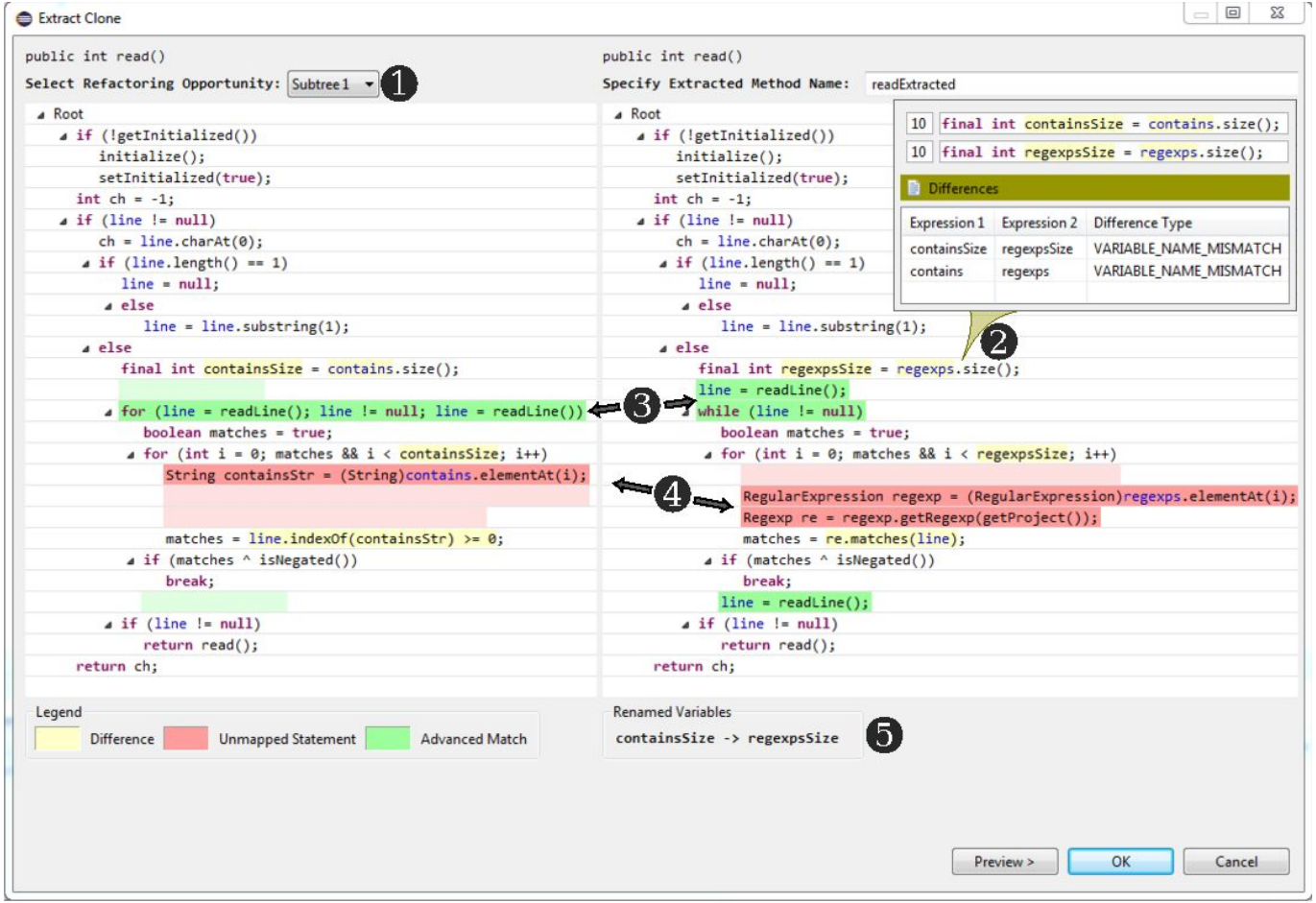
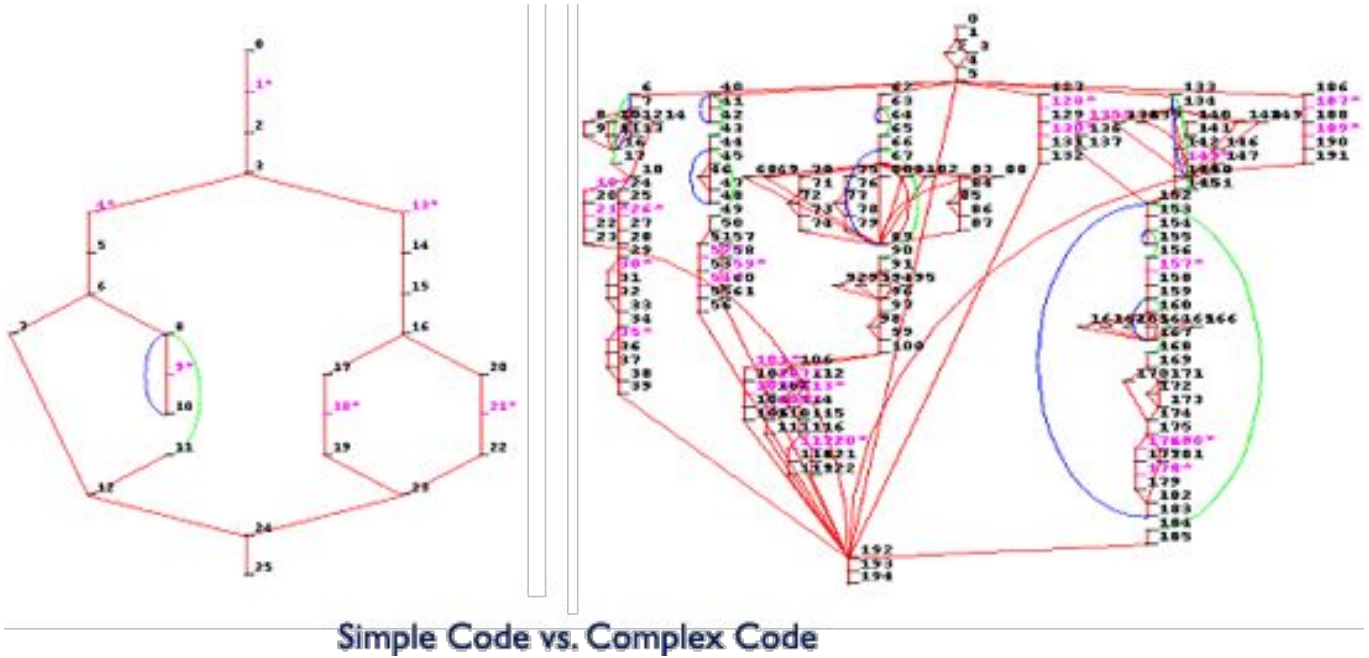


Figure 2: Clone pair visualization and refactorability analysis.

Code Complexity is another source of Technical Debt



Simple Code vs. Complex Code

Code Complexity

Cyclomatic complexity = number of conditions + 1

Minimum number of test cases required to achieve branch coverage

Higher cyclomatic complexity usually, but not always, makes code harder to understand and maintain

Not always?

This code has cyclomatic complexity 14, far beyond the typical threshold - but no developer would consider this code complex

Also need to consider *nesting depth* = maximal number of control structures (loops, if) that are nested inside each other, which probably correlates more closely with understanding and maintenance challenges

```
String getMonthName (int month) {
    switch (month) {
        case 0: return "January";
        case 1: return "February";
        case 2: return "March";
        case 3: return "April";
        case 4: return "May";
        case 5: return "June";
        case 6: return "July";
        case 7: return "August";
        case 8: return "September";
        case 9: return "October";
        case 10: return "November";
        case 11: return "December";
        default: throw new
IllegalArgumentException();
    }
}
```

Both of these examples have cyclomatic complexity 5

```
String getWeight(int i) {
        if (i <= 0) {
            return "no weight";
        }
        if (i < 10) {
            return "light";
        }
        if (i < 20) {
            return "medium";
        }
        if (i < 30) {
            return "heavy";
        }
        return "very heavy";
}
```

```
int sumOfNonPrimes (int limit) {
    int sum = 0;
    OUTER: for (int i = 0; i < limit; ++i) {
                if (i <= 2) {
                        continue;
                }
                for (int j = 2; j < i; ++j)  {
                        if (i % j == 0) {
                                continue OUTER;
                        }
                }
                sum += i;
        }
        return sum;
}
```

Other Technical Debt

Dispensable unused code - sometimes previously used (dead code), sometimes created “just in case” for future (speculative generality)

Don’t just comment out dead code, delete it - if it turns out you need it again, it’s still in the version control repository

Reducing code size (even in comments) means less code to understand and maintain

See Knightmare: A DevOps Cautionary Tale

Code Review

Style checkers and static analysis bug finders cannot find all bad code

In “code review”, one or more *humans* read and analyze the code

Like automated static analysis, code review can potentially consider *all* inputs whereas testing can only consider *some* inputs - often a very small subset of all inputs.

Code review can find problems that dynamic testing and static analyzers cannot, such as code that passes style checking and other static analysis, and passes all test cases, but is still unreadable

Humans reviewers can also spot application-specific bugs as well as generic bugs that are beyond the state of the art in static analysis

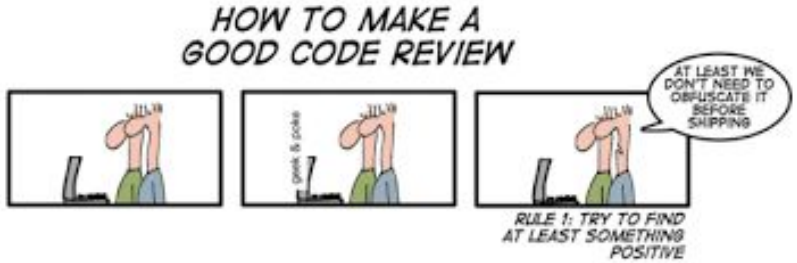
Pair Programming

Instant code review happens on the fly during pair programming, where the navigator reviews while the driver writes/edits

Training

Code review helps train junior developers - over-the-shoulder synchronous review by senior developer

Lightweight Code Review



Many organizations require code to be reviewed by a peer, a senior developer, or a designated reader prior to committing to the shared code repository

Lightweight forms of code review involve one or two readers besides the coder

May be synchronous (reviewer sits with coder immediately after coder “done”) or asynchronous (separately)

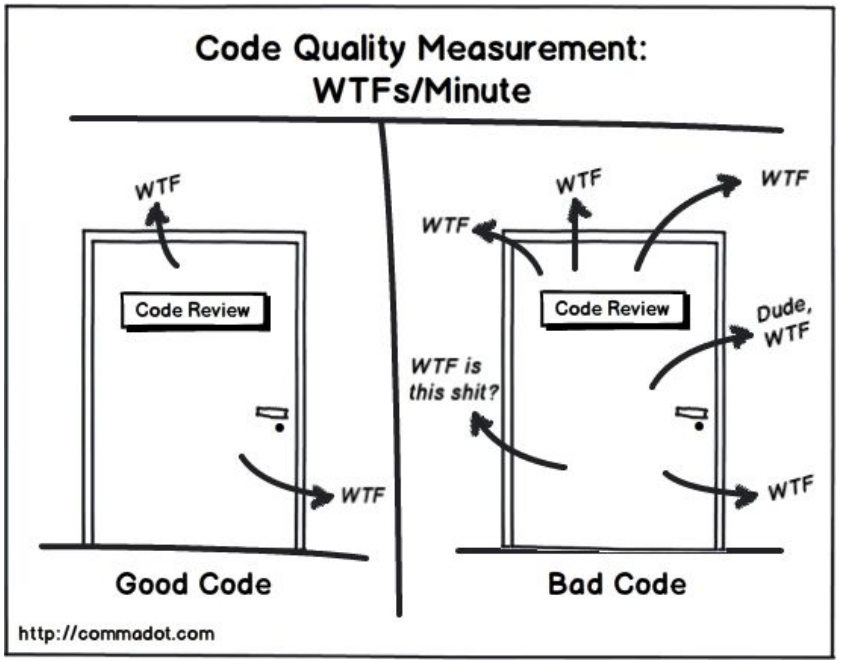
Aside from pair programming and junior developer training, there are major problems with synchronous reviews, so asynchronous generally considered better

Formal Code Review

Formal code review (sometimes called Fagan inspection) is heavyweight - involves a *team* review process

Mandated for safety-critical software, or any software that must work right the first time and every time

Rarely used for conventional business and consumer software other than for training purposes



Formal Code Review Roles

- Moderator - runs meeting, often from outside development team such as another team or a separate quality assurance group
- Recorder - takes notes
- Reader - sometimes author, sometimes intentionally not the author (who may not find code intent as obvious as the author). Reads the code aloud, walkthrough line by line, explain along the way
- Everyone else is called an Inspector - attendees might include customer representatives, end-users, testers, customer/technical support, etc.

Formal Code Review Process

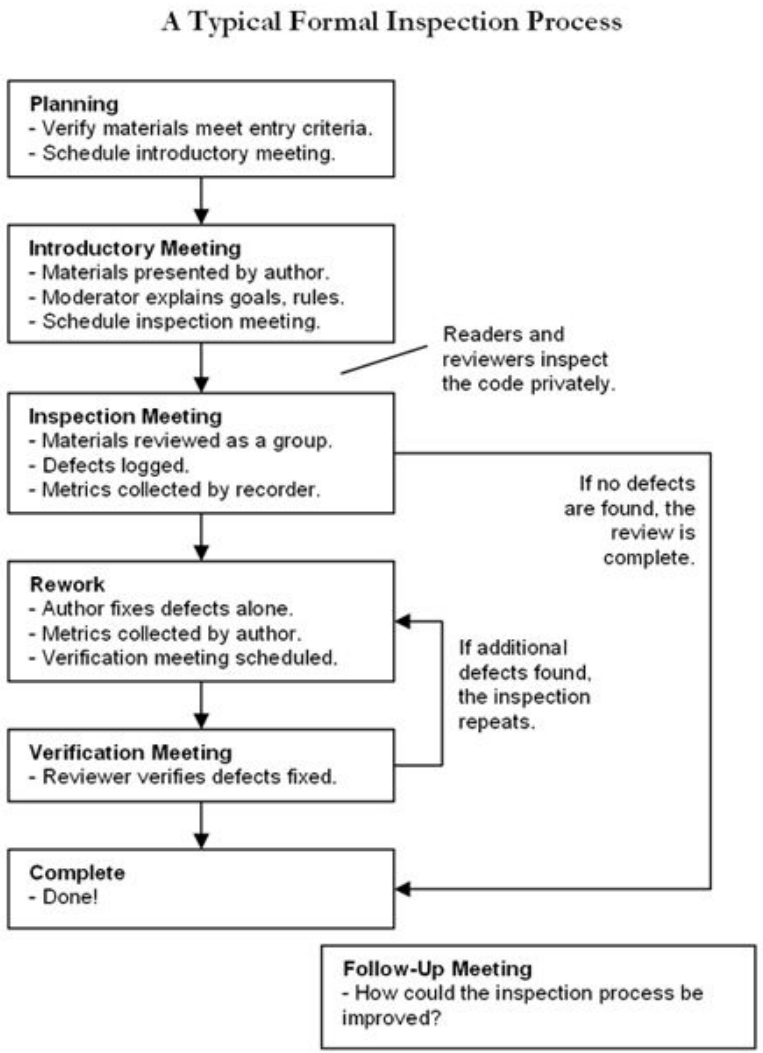
Formal inspection proceeds in several steps

First make should already compiles, passes static analysis and unit tests (don't waste human time for work that can easily be automated)

Check the output of each step and compare it to the desired outcome

Decide whether to move on to the next step or still have work to do in the current step

Meeting <= 2 hours



Team Project

Preliminary Project Proposal due tomorrow

Your service has to implement some API, but it does not need to be a REST API, and could involve interacting with other services, RPC, mediation, callbacks, events, etc. - but no GUI

No GUI does not necessarily mean no UI, there might be an administrative console using CLI (command line interface), just make sure your service cannot be mistaken for an “app”