COMS W4156 Advanced Software Engineering (ASE)

October 13, 2022

Agenda

1. Working in Teams
2. CheckStyle demo
3. Pair Programming

Software Engineers always work in Teams

People who write run-once programs are programmers not software engineers

Teams always have at least three members: you, the past you, and the future you

You / Past You / Future You

Motivate...

version control - It worked last week, why isn’t it working now?

coding style - Why did I name this function ‘FixThisLater’ two years ago?

code documentation - What does ‘FixThisLater’ do?

task board - Where is the sticky note with my to-do list?

....even if your team never grows

Multi-Member Teams

There are usually more members, all of which join the team at some point (not necessarily beginning) and may leave at some point (not necessarily end)

A team of 4-5 software engineers often has >5 other members (some shared with other teams), such as tech lead, engineering manager, product manager, tech writer, customer support and/or devops liaison, UIUX designer and/or researcher (when applicable), possibly a separate testing team, with several layers of management above each

Some teams also have student interns or trainees

Multi-Member Engineering Teams

Mandate...

version control - My code worked last week, now it doesn't, who changed it?

coding style - Why did Jing, who left two years ago to work for Kookle, name this function 'FixThisLater'?

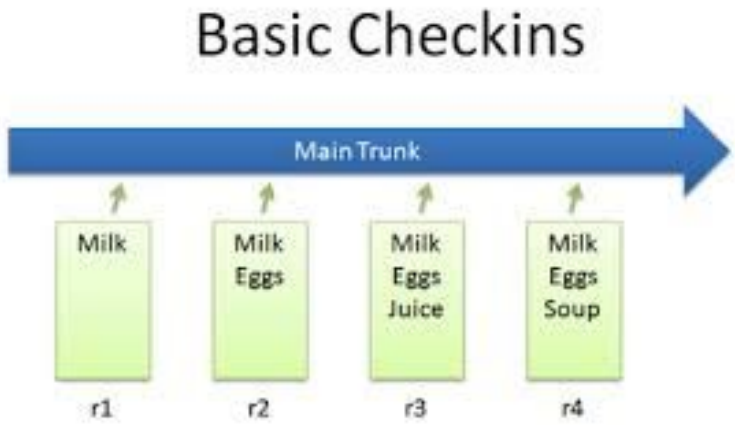
code documentation - What does 'FixThisLater' do?

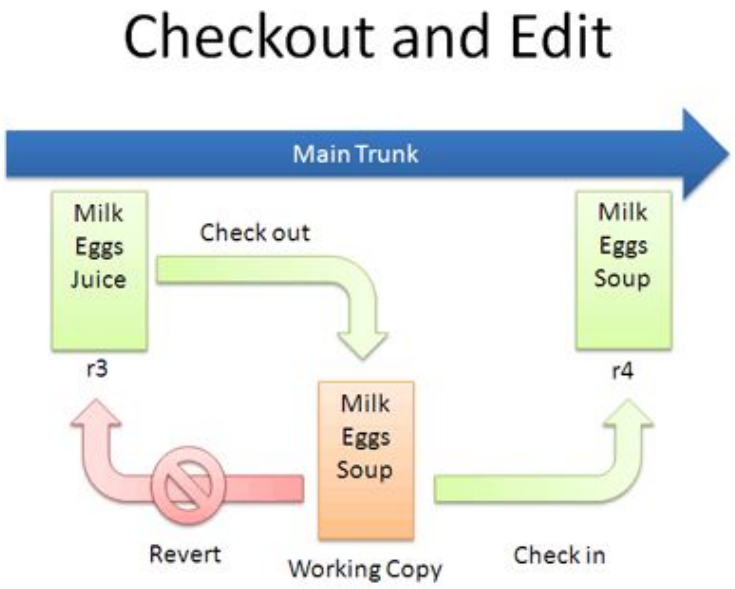
task board - What am I supposed to be working on?

...particularly as team grows or churns

Version Control

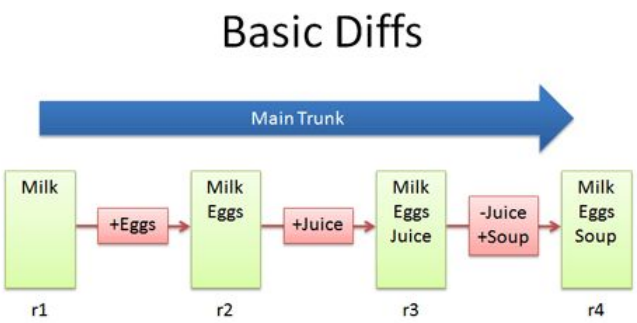
All (or nearly all) of you already understand basic version control

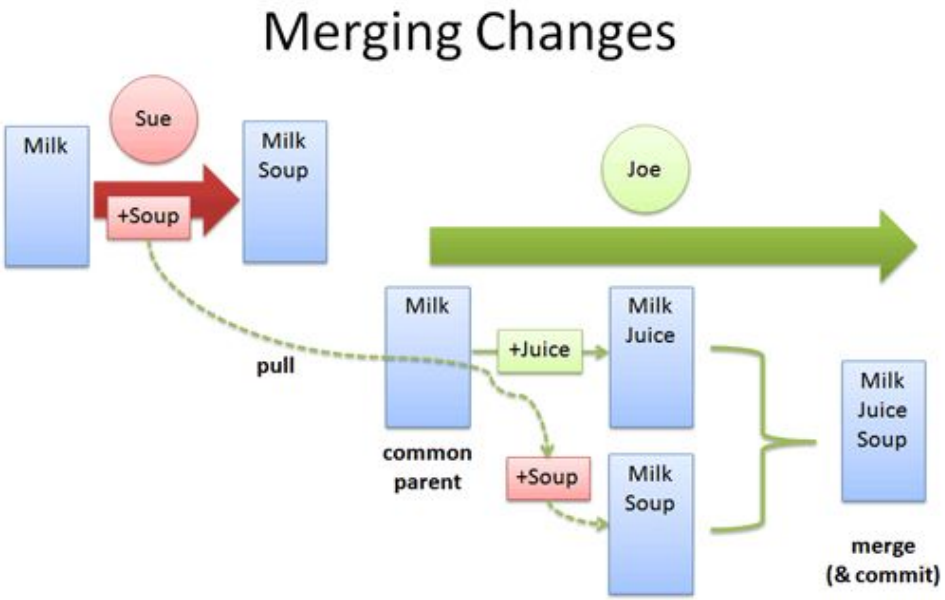




Version Control

But maybe not how merging changes works





Example Auto-Merge

```
int main(){
   char* ptr;
   FILE* data;
   char buf[100];
   ….some code….
   data = fopen("data.txt", "r");
   if(fgets(buf,100,data) == NULL)
      return 1;
   printf("Read: %s", buf);
   fclose(data);
   data = fopen("data.txt", "w");
   fwrite("juice\n", 6, 1, data);
   fclose(data);
   return 0;
   }
```

```
int main(){
   char* ptr;
   FILE* data;
   char buf[100];
   ….some code...
   data = fopen("data.txt", "r");
   if(fgets(buf,100,data) == NULL)
      return 1;
   printf("Read: %s", buf);
   fclose(data);
   data = fopen("data.txt", "w");
   fwrite("soup\n", 7, 1, data);
   fclose(data);
   return 0;
   }
```

Example Probably Needs Manual Resolution

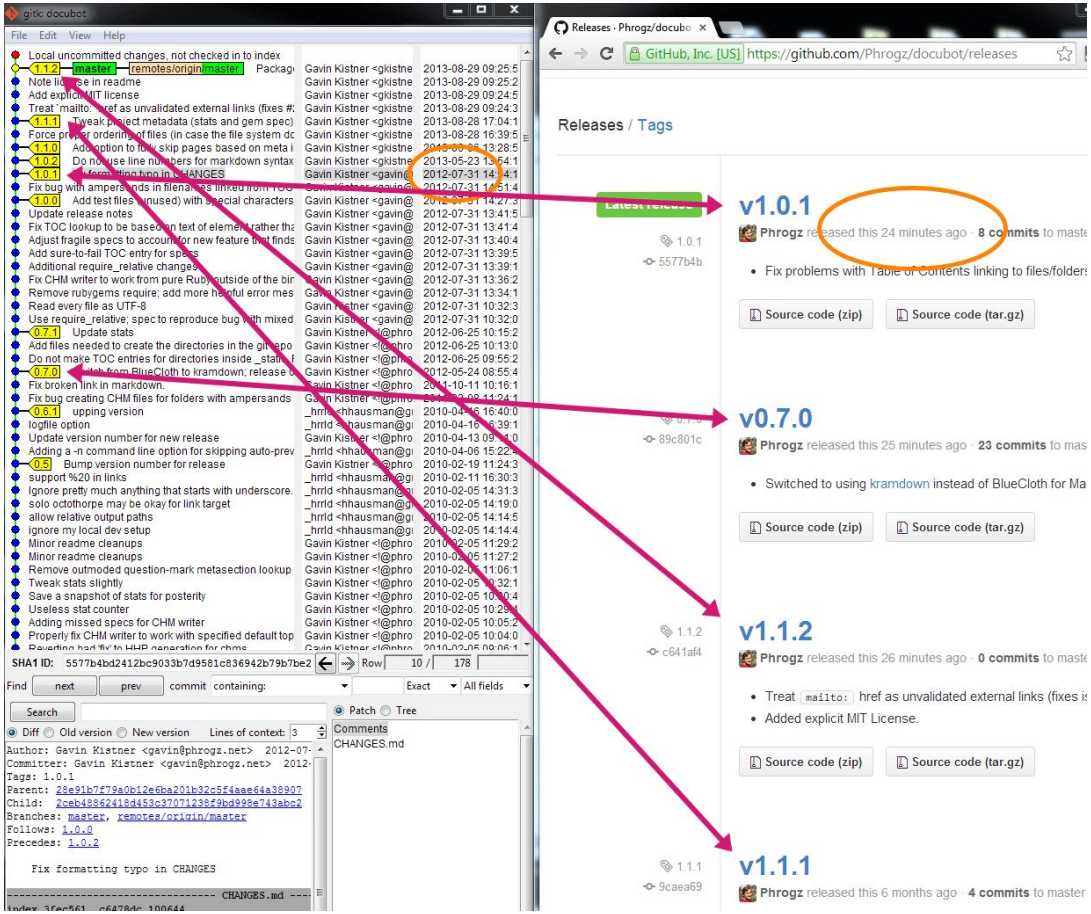
```
/* original */
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* pass = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(pass, "default", 8);
   ...some code...
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

```
/* concurrent change */
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* pass = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(pass, "default", 8);
   …. some code...
   read(0, user, 255);
   read(0, pass, 255);
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

```
/* concurrent change */
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* passwd = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(passwd, "default", 8);
   ...some code...
   read(0, passwd, 255);
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

Release Tags

When you submit an iteration or present a demo, **tag** the revisions that contributed to that milestone so you can return to it later



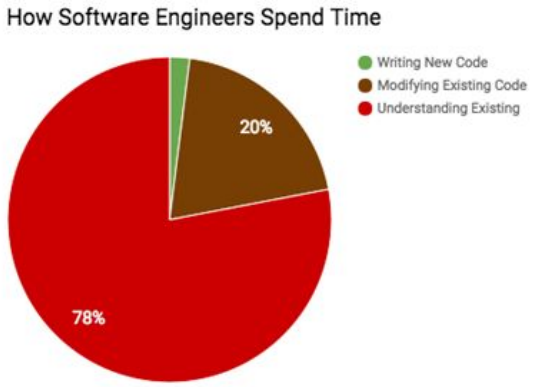
Coding Style

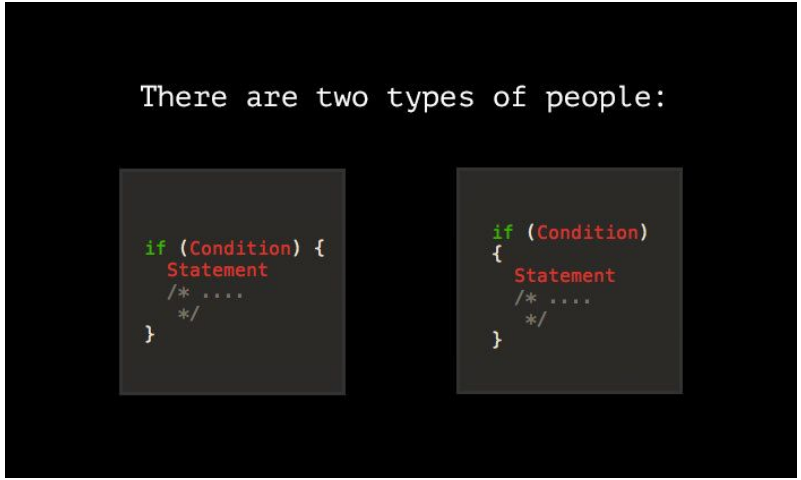
All (or nearly all) of you learned some standard coding style when you learned to program in xxx

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]





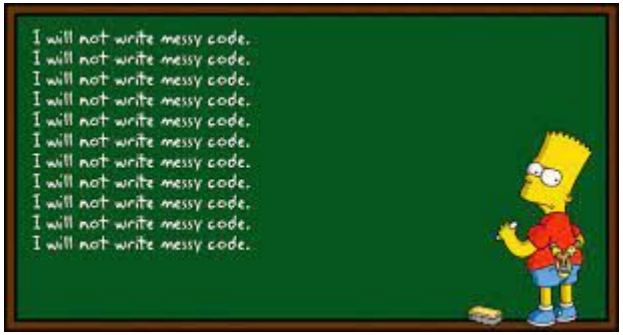
What is Wrong with this Code?

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

    case 2:
        doSomethingElse();
        break;

    default:
        doDefaultThing();
}
```

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

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

Code Documentation

Classes (modules, files, etc.) document the class as a whole, overview what the class does and show 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 can I learn more?

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)

Task Boards

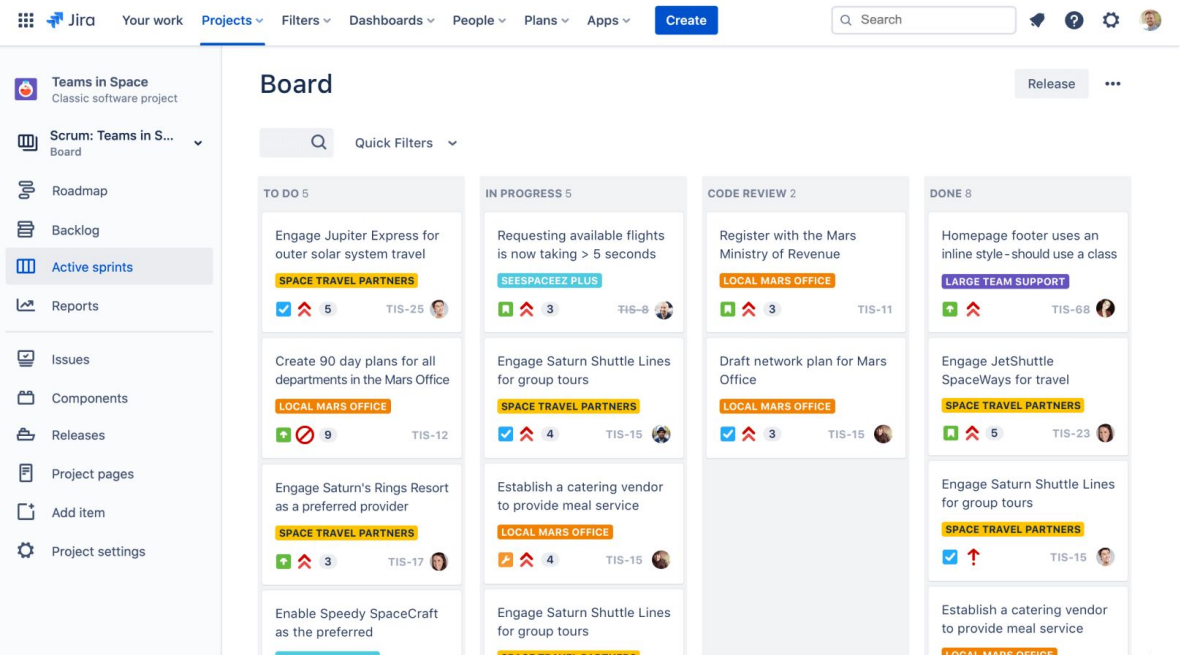
Documents who is working on what this iteration

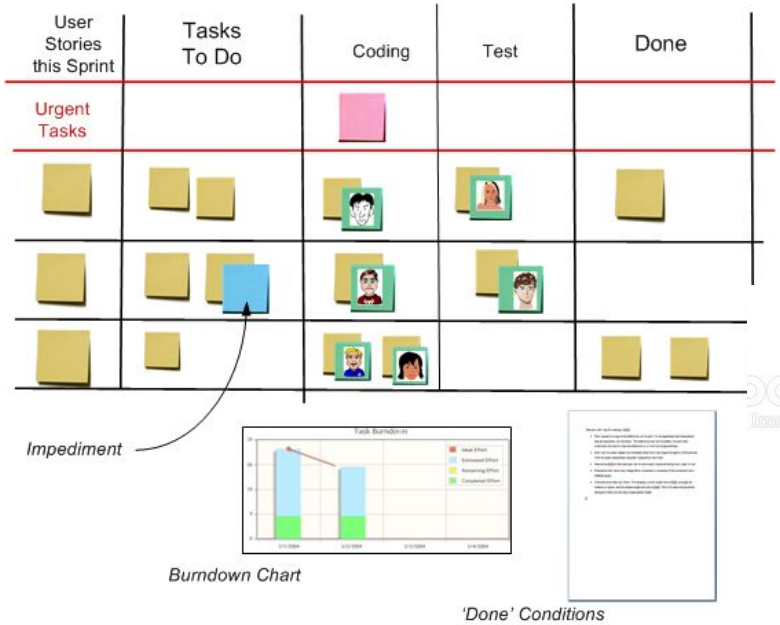
Tasks boards show to-do, in-progress, etc. columns of tickets - each with title, priority, who assigned to, tags, description, project its part of, and links to other tickets, revisions and/or releases

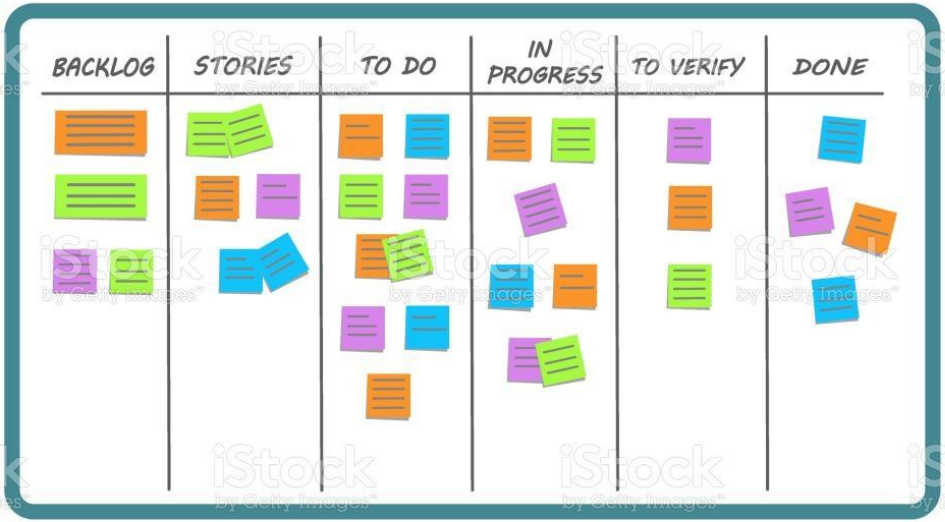
Tickets typically written and assigned by team leader or product owner

JIRA and Trello probably best-known systems, there's free versions for small teams

Github has built-in "issues" and "projects"




User Stories this Sprint
Tasks To Do
Coding
Test
Done
Urgent Tasks
Impediment
Burndown Chart
'Done' Conditions


BACKLOG
STORIES
TO DO
IN PROGRESS
TO VERIFY
DONE

Agenda

1. Working in Teams
2. CheckStyle demo
3. Pair Programming

CheckStyle demo: Ira Ceka

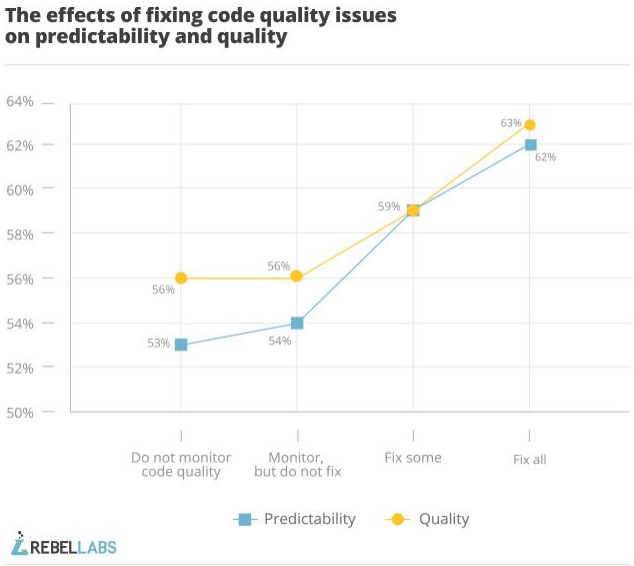
demo presentation

Read the Docs

CheckStyle is for Java

SonarLint is for C++ (and for Java and many other languages)

Also see SonarQube + SonarLint



Agenda

1. Working in Teams
2. CheckStyle demo
3. Pair Programming

Multi-Member Engineering Teams

Enable…

pair programming!

Ought to be called “pair software engineering”, but “pair programming” sounds better

Used for many software engineering activities, not just programming

Some companies use pair-programming interviews between job candidate and evaluator (current employee) even if their employees do not pair-program routinely

Ideal Pair Programming

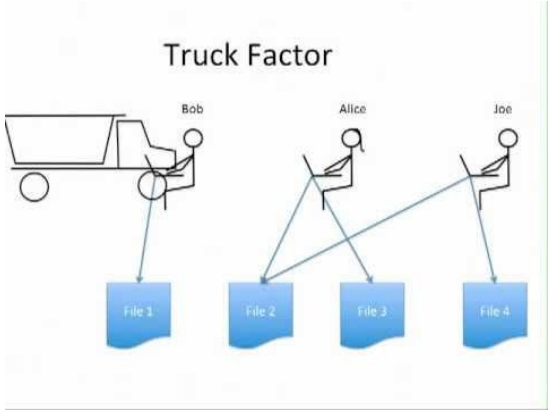
Two people sit side by side at same computer or use remote desktop sharing

- Take turns “driving” (typing) vs. “navigating” (continuous review of code or whatever working on)
- Switch roles at frequent intervals (eg, 25 minutes)
- The pair does **not** divide up the work, they do everything **together**
- If necessary to work separately on something, eg, when one partner is absent, they review together

Benefits of Pair Programming

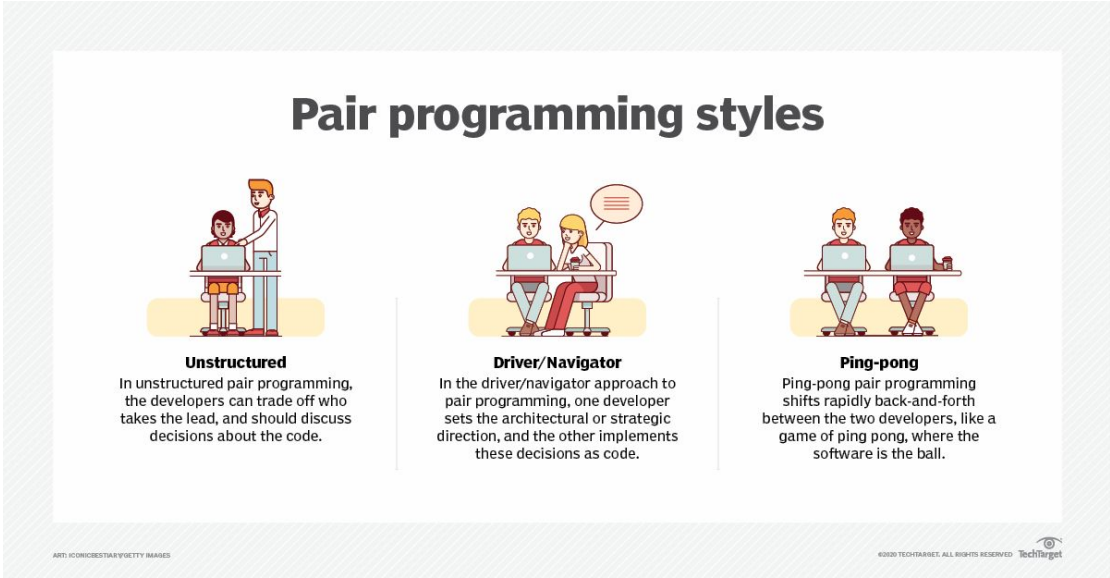
Reduce risk through collective code ownership and maximize “truck factor” = number of engineers that would have to disappear before project would be in serious jeopardy

- Shared responsibility to complete tasks on time
- Stay focused and on task
- Less likely to read email, surf web, etc.
- Less likely to be interrupted
- Partners expect “best practices” from each other
- Two people can solve problems that one couldn’t do alone and/or produce better solutions
- Pool knowledge resources, improve skills



Pair Programming

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



Code Review

Style checkers and static analysis bug finders cannot find all bad code (static analysis bug finders discussed next week)

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 analyses, 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

Training

Pair programming helps train junior developers - over-the-shoulder synchronous code 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)

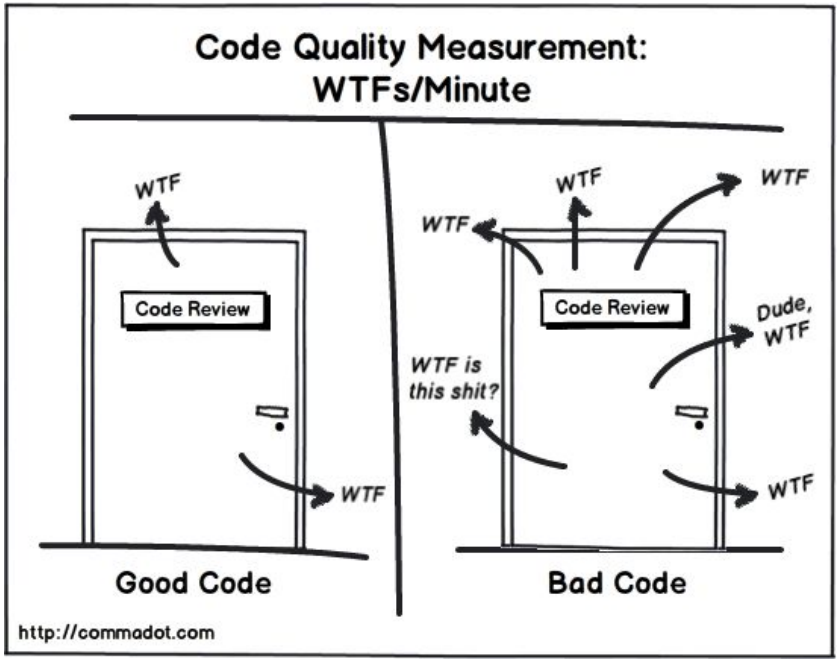
Aside from pair programming and junior developer training, there are [major problems with synchronous reviews](), so asynchronous generally considered better - submit prospective changes (pre-commit) for review and do something else

Formal Code Review

Formal code review (sometimes called Fagan inspection) is heavyweight - involves an elaborate *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



Pair Programming and Code Review for Your Projects

We strongly recommend (but do not require) pair programming for your team projects. This is why four-member teams are preferred to five-member teams, since triplet programming does not work very well - but five member teams can trade off pairs

We strongly recommend (but do not require) lightweight code review for your team projects. Please ask some other member of your team to read your code (or your pair's code) before committing to the main trunk in your github repository, even if it passes all tests and style checking

Upcoming Assignments

First Iteration due October 24

First Iteration Demo due October 31

➔ You can keep coding and testing between first iteration submission and first iteration demo, but tag both separately! Also see github releases

Next Class

multi-client API Testing demo

Finding Bugs: beyond testing

Ask Me Anything