Lecture Notes
September 17, 2020

Working in a team: *Coding Style*

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.

Also known as ‘coding conventions’ or ‘coding standards’.

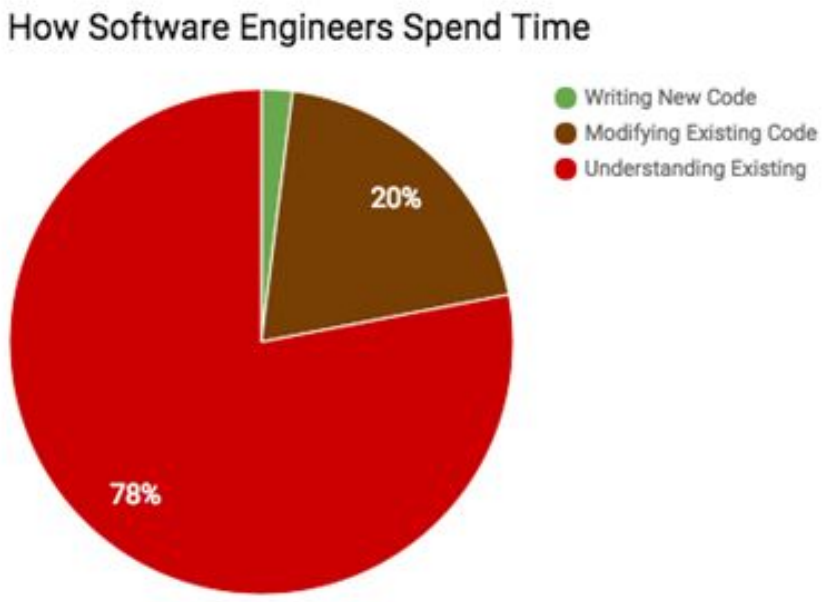
There are coding style guides for nearly every programming language in active use.

See Google’s many style guides.

Software engineering teams should adopt and consistently follow some coding style guide when writing source code, even when working in a team of one member (i.e., alone), to “collaborate” with one’s past and future self.

Coding conventions help other developers understand your code and vice versa. Coding conventions help your future self understand your past self.

Code is read far more often than it is written



See [How To Write Unmaintainable Code].

Read the following Java code and consider which tests pass - neither, first, second, both

```
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);
    }

}
```

Only the first test passes, the second computes 11.

This coding style violation concerns indentation - which may mask a bug. Remember this is Java, not Python.

Coding styles may include patterns to be avoided as well as patterns to be used, make errors more obvious - unfamiliar patterns jump out of the code when looked at

What is wrong with this code?

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

    case 2:
        doSomethingElse();
        break;

    default:
        doDefaultThing();
}
```

Imagine a style guide that says something like *“All switch statement cases must end with break, throw, return, or a comment indicating an intentional fall-through.”*

```
switch(value) {
    case 1:
        doSomething();
        //falls through

    case 2:
        doSomethingElse();
        break;

    default:
        doDefaultThing();
}
```

What does this code do?

Obfuscated C

Spoiler

(See The International Obfuscated C Code Contest.)

Style checker = tool that automatically detects deviations from prescribed coding style.

There are many style checker tools, both IDE or code editor plugins and standalone tools.

The auto-merge feature of git and other VCSs depend on style compliance to avoid superficial, format-related merge conflicts. So should always run the style checker tool and fix any problems before commiting code changes to the version control system.

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

Style checking is a simple form of “static analysis”. Static analysis parses the code like a compiler but instead of producing an executable it produces errors and warnings.

Static analysis includes anything an automated tool can do just by looking at the code, without execution. Static analysis applies to a range of inputs, possibly “all” inputs - or all valid inputs (what are invalid inputs?). Static analysis looks for “patterns” in the code according to certain rules to always do and/or rules to never do.

Dynamic analysis requires execution, besides testing includes “checking”, monitoring and measuring, etc. and applies to only those inputs on which the code executed. This is a reason for monitoring after software has been deployed, to see what happens with many more inputs (from users) than were tested (by developers).

Later on in the course we will look at more sophisticated static analyzers, for finding “code smells” and some common bug patterns, as well as study testing techniques and possibly other dynamic analysis.

You will use the CheckStyle plugin for Eclipse in your individual project. By default, the Checkstyle plugin checks for compliance with the Google Style Guide for Java. It's possible to change the style guide used by the Eclipse CheckStyle plugin (Window->Preferences->Checkstyle) to Sun coding conventions for Java or custom rules.

"Assignment I1":
https://courseworks2.columbia.edu/courses/104335/assignments/482592

To do this assignment, you need to code a simple web application in Java (a tic-tac-toe game) using the Javalin lightweight framework. You need to generate CheckStyle reports for your code and include them in your codebase repository on github. If a CheckStyle report shows problems, you should fix the problems and rerun CheckStyle. Repeat until the CheckStyle report is 'clean' or you are unable to fix the remaining problems (or you run out of time).

Later on, your team project will need to use an analogous style checker tool for your chosen programming language. Then the tool can be either standalone or an addon to your IDE or code editor.

“Assignment T0”: https://courseworks2.columbia.edu/courses/104335/assignments/486855 This is the initial team assignment, to form a team and choose which programming language your team will use (C/C++, Java, Javascript, Python).

It’s not due until October 15, after the individual project and first assessment, but you can start forming teams any time.

There is a “search for teammates” thread in piazza that you can use to look for teammates https://courseworks2.columbia.edu/courses/104335/external_tools/1456

If time permits, at the end of each of the next few class sessions I will put everyone in breakout groups so you can introduce yourself and meet each other. Today (or next class if no time today) there is a short exercise to work on in your breakout groups, and then volunteers will report their group’s ideas to the class:

Recall the small drone that flies around an in-person classroom and uses its sensors to determine which students are “paying attention” and which are not. Its sensors include high-resolution camera, sensitive microphone, wifi monitoring, and anything else you can think of that could realistically fit on a drone.

Devise an ***acceptance testing plan*** for how to test the drone software to make sure it works properly on both common cases and corner cases. If it doesn’t work properly, then it’s not acceptable. Acceptance testing is from the user’s point of view, not the developer’s, and treats the software as a blackbox. The users may be represented by human technical staff who intentionally try to confuse the drone.

- The drone software should make best effort to avoid reporting ***false negatives*** = fails to report a student who is not paying attention.
- And make best effort to avoid reporting ***false positives*** = reports student as not paying attention when actually she is paying attention.

Please ***return to the main room*** to report after the breakout rooms close (after 10 minutes if they don’t close).

Addendum:

Ideally, testing has a “test oracle” to determine whether the results of each test is right or wrong. The test oracle may be automated (e.g., assertions) or human judgment. Finding a test oracle may be hard, or impossible, for some software or parts of some software. There will be a lot of discussion about test oracles later in the course.

Acceptance testing is different from beta testing. When representatives of the user organization participate in acceptance testing, their goal is to test the software to check that it is acceptable - it fulfills their needs.

During *beta testing*, users have agreed to try out the software while they go about their normal activities. Their main goal is to accomplish their activities, not to test.

For your team projects, you will need to devise acceptance tests. We do not expect you to recruit users to beta test, but if you do that’s great!