L3: Code Review

Today
- Code reviewing in practice
- Style vs. substance
- Code smells

Recap: the Big Three Properties
Let’s review the three properties of good software that we’re going to focus on the most in this class:

- **Safe from bugs.** Correctness (correct behavior right now), and defensiveness (correct behavior in the future).
- **Easy to understand.** Has to communicate to future programmers who need to understand it and make changes in it later (including yourself)
- **Ready for change.** Software always changes. Some designs make it easy to make changes; others require throwing away and rewriting a lot of code.

Everything we’ll talk about today is intended to improve one or more of these good properties.

Code Reviewing

- **other eyes looking at the code can find bugs**
  - code review is careful, systematic study of source code by others (not original author)
  - analogous to proofreading an English paper
  - look for bugs, poor style, design problems, etc.
  - formal inspection: several people read code separately, then meet to discuss it
  - lightweight methods: over-the-shoulder walkthrough, or by email
  - many dev groups require a code review before commit

- **code review complements other techniques**
  - code reviews can find many bugs cheaply
  - also test the understandability and maintainability of the code

Code reviewing is widely practiced in open source (e.g. Mozilla project)

Code reviewing is widely practiced in industry: at Google, you can’t push any code until another engineer has signed off on it in a code review. (“LGTM” = “looks good to me”)

How we’ll do code reviewing: see the Code Reviewing document on Stellar
Style Standards

Most companies and large projects have coding style standards. These can get pretty detailed, even to the point of specifying: whitespace (how deep to indent) and where curly braces and parentheses should go. These kinds of questions often lead to holy wars (http://www.outpost9.com/reference/jargon/jargon_23.html#TAG897) since they end up being a matter of taste and style.

For Java, there’s a general style guide at http://www.oracle.com/technetwork/java/codeconv-138413.html. Some of its advice gets very specific:

- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

In 6.005, we have no official style guide of this sort. We’re not going to tell you where to put your curly braces. That’s a personal decision that each programmer should make. It’s important to be self-consistent, however. It’s also important to follow the conventions of the project you’re working on. If you’re the guy who reformats every module you touch to match your personal style, your teammates will hate you, and rightly so. Be a team player.

In 6.005, we would rather discuss the reasons for and against various rules, to help you understand and appreciate those rules when you have to follow them in the future.

For example, let’s discuss this possible rule:

- Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

The Rest of This Lecture Is Not Exhaustive

But there are some rules that are quite sensible and target our big three properties, in a stronger way than placing curly braces.

The rest of this lecture talks about some of these rules, at least the ones that are relevant at this point in the course, where we’re mostly talking about writing basic Java.

These are some things you should start to look for when you’re code reviewing other students, and when you’re looking at your own code for improvement.

Don’t consider it an exhaustive list of code style guidelines, however. Over the course of the semester, we’ll talk about a lot more things – specifications, abstract data types with representation invariants, concurrency and thread safety – which will then become fodder for code review.

Smelly Example #1

Programmers often describe bad code as having a “bad smell” that needs to be removed. “Code hygiene” is another word for this. Let’s start with some smelly code.

```java
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 120;
    } else if (month == 6) {
        dayOfMonth += 151;
    } else if (month == 7) {
        dayOfMonth += 182;
    } else if (month == 8) {
        dayOfMonth += 213;
    } else if (month == 9) {
        dayOfMonth += 244;
    } else if (month == 10) {
        dayOfMonth += 274;
    } else if (month == 11) {
        dayOfMonth += 305;
    } else if (month == 12) {
        dayOfMonth += 336;
    }

    return dayOfMonth;
}
```
Don’t Repeat Yourself

Duplicated code is a risk to safety. If you have identical or very-similar code in two places, then the fundamental risk is that there’s a bug in both copies, and some maintainer fixes the bug in one place but not the other.

Avoid duplication like you’d avoid crossing the street without looking. Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the block you’re copying, the riskier it is.

The dayOfYear example is full of identical code. How would you remove it?

Comments Where Needed

A quick general word about commenting. Good software developers write comments in their code, and do it judiciously. Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change.

One kind of crucial comment is a specification, which appears above a method or above a class and documents the behavior of the method or class. In Java, this is conventionally written as a Javadoc comment, meaning that it starts with /** and includes @-syntax (like @param and @return for methods.) Here’s an example of a spec:

```java
/**
 * Compute the hailstone sequence.
 * See
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 * For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) { return ... 
```
Specifications document assumptions. More to come in the next few lectures.

Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere.

**Some comments are bad and unnecessary.** Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java:

```java
while (n != 1) { // test whether n is 1  don’t do this!
    ++i; // increment i
    l.add(n); // add n to l
}
```

But obscure code _should_ get a comment:

```java
sendMessage("as you wish"); // this basically says "I love you"
```

The dayOfYear code needs some comments – where would you put them? For example, where would document whether month runs from 0 to 11 or from 1 to 12?

**Fail Fast**

dayOfYear doesn’t fail fast – if you pass it the arguments in the wrong order, it will quietly return the wrong answer. In fact, the way dayOfYear is designed, it’s highly likely that a non-American will pass the arguments in the wrong order!

It needs more checking – either static checking or dynamic checking.

**Avoid Magic Numbers**

There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2. Okay, three.

Other constant numbers need to be explained. One way to explain them is with a comment, but a far better way is to _declare the number as a constant with a good, explanatory name_.

Magic numbers are bare should be declared as constants with readable, descriptive names. dayOfYear is full of magic numbers:

- the months 2, ..., 12 would be far more readable as FEBRUARY, ..., DECEMBER
- the days-of-months 30, 31, 28 would be more readable (and eliminate duplicate code) if they were in a data structure like an array, list or map, e.g. `MONTH_LENGTH[month]`
- the mysterious numbers 59 and 90 are particularly pernicious examples of magic numbers. Not only are they uncommented and undocumented, they are actually the result of a _computation done by hand_ by the programmer. Don’t hardcode constants that you’ve computed by hand. Java is better at arithmetic than you are. Explicit computations like “31 + 28” make the provenance of these mysterious numbers much clearer. “MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]” would be clearer still.
Have One Purpose For Each Variable

In the dayOfYear example, the parameter dayOfMonth is reused to compute a very different value – the return value of the function, which is not the day of the month.

Don't reuse parameters, and don't reuse variables. Variables are not a scarce resource in programming. Mint them freely, give them good names, and just stop using them when you don’t need them. You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.

Not only is this an ease-of-understanding question, but it’s also a safety-from-bugs and ready-for-change question.

Method parameters, in particular, should generally be left unmodified. (This is important for being ready-for-change – in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn't blow them away while you’re computing.)

It’s a good idea to use final for method parameters, and as many other variables as you can. Final says that the variable should never be reassigned, and the Java compiler will check it statically. For example:

```java
public static int dayOfYear(final int month, final int dayOfMonth, final int year) { ... }
```

Smelly Example #2

There was a latent bug in dayOfYear. It didn’t handle leap years at all. As part of fixing that, suppose we write a leap-year test method.

```java
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == '5'
    || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3) == '2' || tmp.charAt(3) == '6')
            return true;
        else
            return false;
    }
    else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false;
        }
        if (tmp.charAt(3) == '0' || tmp.charAt(3) == '4' || tmp.charAt(3) == '8')
            return true;
    }
    return false;
}
```

What are the bugs hidden in this code? And what style problems that we’ve already talked about?

Use Good Names

Good method and variable names are long and self-descriptive. Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables. For example, you can rewrite

```java
int tmp = 86400; // tmp is the number of seconds in a day don’t do this!
```
int secondsPerDay = 86400;

tmp and temp are awful variable names, symptoms of extreme programmer laziness. Every local variable is temporary! Better to use a longer, more descriptive name, so that your code reads clearly all by itself.

Follow the lexical naming conventions of the language. In Python, classes are typically Capitalized, variables are lowercase, and words are separated by underscores. In Java:

- methodsAreNamedWithCamelCaseLikeThis
- variablesAreAlsoCamelCase
- CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES
- ClassesAreCapitalized
- packages are lowercase and separated by dots

The leap method has bad names: the method name itself, and the local variable name.

**Smelly Example #3**

Here’s a third example of smelly code.

```java
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

**Don’t Use Global Variables**

Let’s parse out global variable. A global variable means two things:

- a variable (a name whose meaning can be changed)
  - that is global (accessible and changeable from anywhere in the program)

The countLongWords uses two global variables:

```java
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;
```

Why are global variables dangerous? Which of these could be made into a constant instead, and how would you do that?

In general, change global variables into parameters and return values, or into objects that you’re calling methods on.
Coherent Methods

countLongWords isn’t coherent – it does two different things, counting words and finding the longest word. Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change).

Methods Should Return Results, not Print Them

countLongWords isn’t ready for change. It sends some of its result to the console, System.out. That means that if you want to use it in another context – where the number is needed for some other purpose, like computation rather than human eyes – it would have to be rewritten.

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results. (The sole exception here is debugging output, which can of course be printed to the console, but that shouldn’t be a part of your design, only a part of how you debug your design.)

Use Whitespace to Help the Reader

Use consistent indentation. The leap example is bad at this. The dayOfYear example is much better. In fact, dayOfYear nicely lines up all the numbers into columns, making them easy for a human reader to compare and check. That’s a great use of whitespace.

Put spaces within code lines to make them easy to read. The leap example has some lines that are packed together – put in some spaces.

Never use tab characters for indentation, only space characters. Note that we say characters. We’re not saying you should never press the Tab key, only that your editor should never put a tab character into your source file in response to your pressing the Tab key. The reason for this rule is that different tools treat tab characters differently – sometimes expanding them to 4 spaces, sometimes to 2 spaces, sometimes to 8. If you run “git diff” on the command line, or if you view your source code in a different editor, then the indentation may be completely screwed up. Just use spaces. Always set your programming editor to insert space characters when you press the Tab key.

Summary

So far we’ve talked about two techniques for reducing bugs:

- static checking
- code reviews

What are the pros and cons of each one?