Source: https://web.mit.edu/6.005/www/fa15/classes/04-code-review/

Code Review

You must complete the ***reading exercises*** in this reading by 10:00 pm the night before class. Don't forget to log in using the big red ***Log in*** button. You will only receive credit for reading exercises if you are logged in when you do them.

Our Prime Objective in 6.005

Learning to write code that is:

Safe from bugs	**Easy to understand**	**Ready**
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Design rewriti

Objectives for Today's Class

In today's class, we will practice:

- code review: reading and discussing code written by somebody else
- general principles of good coding: things you can look for in every code review, regardless of programming language or program purpose

Code Review

Code review is careful, systematic study of source code by people who are not the original author of the code. It's analogous to proofreading a term paper.

Code review really has two purposes:

- **Improving the code.** Finding bugs, anticipating possible bugs, checking the clarity of the code, and checking for consistency with the project's style standards.
- **Improving the programmer.** Code review is an important way that programmers learn and teach each other, about new language features, changes in the design of the project or its coding standards, and new techniques. In open source projects, particularly, much conversation happens in the context of code reviews.

Code review is widely practiced in open source projects like Apache and Mozilla. Code review is also widely practiced in industry. At Google, you can't push any code into the main repository until another engineer has signed off on it in a code review.

In 6.005, we'll do code review on problem sets, as described in the Code Reviewing document on the course website.

Style Standards

Most companies and large projects have coding style standards (for example, Google Java Style). 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 since they end up being a matter of taste and style.

For Java, there's a general style guide (unfortunately not updated for the latest versions of Java). 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, and it's *very* important to follow the conventions of the project you're working on. If you're the programmer who reformats every module you touch to match your personal style, your teammates will hate you, and rightly so. Be a team player.

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 reading 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.

```
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 += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

The next few sections and exercises will pick out the particular smells in this code example.

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.

Don't Repeat Yourself, or DRY for short, has become a programmer's mantra.

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

READING EXERCISES

Don't repeat yourself

Don't repeat yourself

Don't repeat yourself

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:

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz_conjecture#Statement_of_the_problem
 * @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) {
...
```

}

Specifications document assumptions. We've already mentioned specs a few times, and there will be much more to say about them in a future reading.

Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere. This is vitally important for practicing software developers, and is required by the 6.005 collaboration policy when you adapt code you found on the web. Here is an example:

```
// read a web page into a string

// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code

String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDelimiter("\\A").next();
```

One reason for documenting sources is to avoid violations of copyright. Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive. Another reason for documenting sources is that the code can fall out of date; the Stack Overflow answer from which this code came has evolved significantly in the years since it was first answered.

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:

```
while (n != 1) { // test whether n is 1   (don't write comments like this!)

  ++i; // increment i

  l.add(n); // add n to l

}
```

But obscure code should get a comment:

```
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 you document whether month runs from 0 to 11 or from 1 to 12?

READING EXERCISES

Comments where needed

Fail Fast

Failing fast means that code should reveal its bugs as early as possible. The earlier a problem is observed (the closer to its cause), the easier it is to find and fix. As we saw in the first reading, static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

The dayOfYear function 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.

READING EXERCISES

Fail fast

Fail faster

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 constants.)

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.

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.

READING EXERCISES

Avoid magic numbers

What happens when you assume

Names instead of numbers

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. Introduce them freely, give them good names, and just stop using them when you stop needing 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. The final keyword says that the variable should never be reassigned, and the Java compiler will check it statically. For example:

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 method.

```
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; /*R1*/

      else

         return false; /*R2*/

   }else{

      if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {

         return false; /*R3*/

      }

      if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true; /*R4*/

   }
```

```
    return false; /*R5*/
}
```

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

READING EXERCISES

Mental execution 2016

Mental execution 2017

Mental execution 2050

Mental execution 10016

Mental execution 916

Magic numbers

DRYing out

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

```
int tmp = 86400;  // tmp is the number of seconds in a day (don't do this!)
```

as:

```
int secondsPerDay = 86400;
```

In general, variable names like tmp, temp, and data are awful, symptoms of extreme programmer laziness. Every local variable is temporary, and every variable is data, so those names are generally meaningless. 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

Method names are usually verb phrases, like getDate or isUpperCase, while variable and class names are usually noun phrases. Choose short words, and be concise, but avoid abbreviations. For example, message is clearer than msg, and word is so much better than wd. Keep in mind that many of your teammates in class and in the real world will not be native English speakers, and abbreviations can be even harder for non-native speakers.

The leap method has bad names: the method name itself, and the local variable name. What would you call them instead?

READING EXERCISES

Better method names

Better variable names

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*, not keys. 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.

Smelly Example #3

Here's a third example of smelly code that will illustrate the remaining points of this reading.

```
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

Avoid global variables. Let's break down what we mean by *global variable*. A global variable is:

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

Why Global Variables Are Bad has a good list of the dangers of global variables.

In Java, a global variable is declared public static. The public modifier makes it accessible anywhere, and static means there is a single instance of the variable.

In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on. We'll see many techniques for doing that in future readings.

READING EXERCISES

Identifying global variables

Effect of final

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 kind of output shouldn't be a part of your design, only a part of how you debug your design.

Summary

Code review is a widely-used technique for improving software quality by human inspection. Code review can detect many kinds of problems in code, but as a starter, this reading talked about these general principles of good code:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Return results, don't print them
- Use whitespace for readability

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** In general, code review uses human reviewers to find bugs. DRY code lets you fix a bug in only one place, without fear that it has propagated elsewhere. Commenting your assumptions clearly makes it less likely that another programmer will introduce a bug. The Fail Fast principle detects bugs as early as possible. Avoiding global variables makes it easier to localize bugs related to variable values, since non-global variables can be changed in only limited places in the code.
- **Easy to understand.** Code review is really the only way to find obscure or confusing code, because other people are reading it and trying to understand it. Using judicious comments, avoiding magic numbers, keeping one purpose for each variable, using good names, and using whitespace well can all improve the understandability of code.
- **Ready for change.** Code review helps here when it's done by experienced software developers who can anticipate what might change and suggest ways to guard against it. DRY code is more ready for change, because a change only needs to be made in one place. Returning results instead of printing them makes it easier to adapt the code to a new purpose.