Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 26 — March 25, 2019

Patrick Lam *version 1*

Code Review

Code review is a powerful tool for improving code quality. Today's lecture is based on the following references:

- course reading on code review (main source):

  `http://web.mit.edu/6.031/www/sp17/classes/04-code-review/`
- how code review works in an MIT course on software construction:

  `http://web.mit.edu/6.031/www/sp17/general/code-review.html`
- Fog Creek code review checklist:

  `https://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/`

Our course includes code review as part of Assignment 3. The MIT offering is more comprehensive and requires students to respond to code reviews as well.

Code review is a communication-intensive activity. A reviewer needs to 1) read someone else's code and 2) communicate suggestions to the author of that code. We sometimes think that communicating with the computer is our primary goal when programming, but communicating with other people is at least as important over the long run.

Purpose of code review. The communication inherent in code review aims to improve both the code itself as well as the author of the code. Good code review can give timely information to developers about the context in which their code operates, particularly the project and best-practices uses of the language.

We'll continue with a list of items that you are inspecting when you do a code review.

Formatting. Consistency in formatting helps avoid preventable errors. Positioning of { }s isn't something that necessarily has one right answer. Spaces are probably better than tabs. But the most important thing is to be self-consistent with yourself and within your project.

Code smell example 1.

The following code has a number of bad smells:

```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 += 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;
    }
    // ... through month == 12
    return dayOfMonth;
}
```

Let's go through some of the bad smells.

- **Don't Repeat Yourself**. A few weeks ago in Lecture 23, we talked about code cloning and how it's not always bad. Sometimes it is bad, as in the code above. The usual reason for it being bad is that fixes in one place may remain unfixed in the other place. (Recall: it wasn't always bad when used for forking and templating). For instance, if February actually had 30 days, you'd need to change a lot of code.
- **Fail Fast.** In the language of 6.031, we mean that a defect should be caught closest to when it's written. Static checks, as performed in compilers, catch defects earlier than dynamic checks, which catch defects earlier than letting wrong values percolate in the program state. In this particular example, there are no checks ensuring that a user had not permuted `month` and `dayOfMonth`.
- **Avoid Magic Numbers.** The above code is full of magic numbers. Particularly magical numbers include the `59` and `90` examples, as well as the month lengths and the month numbers. Instead, use names like `FEBRUARY` etc. Enums are a good way to encode months, and days-of-months should be in an array. The `59` should really be `31 + 28`, or better yet, `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]`.
- **One Purpose Per Variable.** The specific variable that's being re-used in the above example is `dayOfMonth`, but this also applies to variables that you might use in your method. Use different variables for different purposes. They don't cost anything. Best to make method parameters `final` and hence non-modifiable.

Comments and code documentation

Code should, ideally, be self-documenting, with good names for classes, methods, and variables. Methods should come with specifications in the form of Javadoc comments, e.g.

```java
/**
 * 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) {
    ...
}
```

Note how this comment describes what the method does, in one sentence, provides context, and then describes the parameters and return values.

Also, when you incorporate code from other sources, cite the sources. For instance, in the A2 `index.html` file:

```
    // adapted from Eli Bendersky's Lexer: http://eli.thegreenplace.net/2013/07/16/hand-written-lexer-in-javascript-compared-to-the-regex-based-ones
    // public domain according to author
    // modifications by Patrick Lam
```

This helps, for instance, when the source is later updated, and is the right thing to do in terms of IP (assuming, of course, that your use of the software is allowed by its license).

Don't write comments that don't contribute to code understanding. If it's blatantly obvious from the code, it shouldn't be a comment. Such comments can mislead and hence do more harm than good.