Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 6 — January 23, 2026

Patrick Lam *version 1*

Code Review

Code review is a powerful tool for improving code quality. Quality Assurance is in the course name, so we'll talk about code review. 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`
- blog post by Vadim Kravcenko on code review:
  `https://vadimkravcenko.com/shorts/code-reviews/`
- Fog Creek code review checklist (archived):
  `https://github.com/godber/software_dev_templates_and_checklists/blob/master/code_review/fog_creek-code_review-checklist.md`

The MIT course has more comprehensive coverage of code reviews; it 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.

From the Kravcenko blog post:

> The real problem was that no one could understand what the hell anyone else had written; we had duplicate logic in many places and different code styles in our modules. It was really bad.

Code review: purpose and levels. An old comment on Hacker News[1] says:

> From my experience doing reviews and having my code reviewed the most important thing is to understand:
>
> - What the intention of change is (what is this trying to achieve)
> - What is the code actually doing
>
> There are three levels of review I typically observe:

[1] `https://news.ycombinator.com/item?id=8862602`

1. Skim, find one or two comments to leave. LGTM!
2. Review each line or method in isolation. Recognize style and formatting errors. Identify a couple local bugs.
3. Understand the purpose and the code.

#3 takes time. It's what it takes however to find the issues where the implementation doesn't actually accomplish what was set out to do (at least in all cases). To recognize where the code is going to break when integrated with other modules. To suggest big simplifications.

#3 is also where you get one of the biggest review benefits—shared understanding of the code base.

Of course, it depends on how big the patch being reviewed actually is. If it's really simple[2], #1 may well be enough. A more in-depth change merits more in-depth 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.

Receiving feedback. We usually talk about giving feedback, not receiving feedback, but receiving is important as well. You don't have to argue about it (or many other things). In person, let the person finish talking. In the context of code review, Kravcenko points out the misconception "So if the code is bad = they are bad", which is of course not true, but it can feel that way. The other misconception is an "us vs them" mentality. You write code, you review code, you are working together to improve code quality.

We'll continue with a list of items that you are inspecting when you do a code review. It is worthwhile to use a checklist customized for your (organization's) needs.

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. Use a tool to check this (and anything else you can check automatically).

Code smell example 1.

The following code has a number of bad smells:

```
1 public static int dayOfYear(int month, int dayOfMonth, int year) {
2     if (month == 2) {
3         dayOfMonth += 31;
4     } else if (month == 3) {
5         dayOfMonth += 59;
6     } else if (month == 4) {
7         dayOfMonth += 90;
```

[2] e.g. https://github.com/Piwigo/Piwigo/commit/c614efd33c696062aca57d5676052f4b7ab85946

```
8       } else if (month == 5) {
9           dayOfMonth += 31 + 28 + 31 + 30;
10      } else if (month == 6) {
11          dayOfMonth += 31 + 28 + 31 + 30 + 31;
12      } else if (month == 7) {
13          dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
14      }
15      // ... through month == 12
16      return dayOfMonth;
17  }
```

Let's go through some of the bad smells.

- **Don't Repeat Yourself**. Code cloning isn't 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, as happened in your Assignment 1 code. (not 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 MIT course 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.

```
1  /**
2   * Compute the hailstone sequence.
3   * See http://en.wikipedia.org/wiki/Collatz_conjecture#
        Statement_of_the_problem
4   * @param n starting number of sequence; requires n > 0.
5   * @return the hailstone sequence starting at n and ending with 1.
6   *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
7   */
```

```
8  public static List<Integer> hailstoneSequence(int n) {
9      ...
10 }
```

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 a previous year's assignment's `index.html` file:

```
1      // adapted from Eli Bendersky's Lexer: http://eli.thegreenplace.net
          /2013/07/16/hand-written-lexer-in-javascript-compared-to-the-
          regex-based-ones
2      // public domain according to author
3      // 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.

Providing feedback. I'm pretty convinced that, as humans, we would prefer not to receive feedback. Certainly it's true for me. Kravcenko writes "I've seen quite a few times when the developers were demotivated for weeks after a bad review. And that's not how it should go." And yet, there are better ways and worse ways to provide feedback; Kravcenko advocates for Nonviolent Code Review[3].

Specifically:

- it's about the code, not the coder
- no personal feelings (on both sides)
- write positive comments, aiming to improve the code

There is an example of using I-messages, like "I'm finding it hard to understand what's happening here." rather than "You write code like a toddler". And ask questions rather than making declarations.

There is one final comment in the Kravcenko post, which is that a code review should take 30 to 60 minutes. If code reviews are taking longer, the pull requests should be smaller.

Static code analysis: PMD

Static analysis is where a tool inspects (most often) the source code of a system and reaches conclusions on possible behaviours or code defects. Contrast that with dynamic analysis, which we've

[3] `https://www.mediawiki.org/wiki/Nonviolent_Code_Review`

seen, where a tool observes system properties from (dynamic) runs of the system, e.g. measuring coverage.

We'll dive into static code analysis by talking about one particular code analysis tool, PMD[4]. Just above, we said that it was better to use tools to flag style issues. PMD is one way to do so. Gitlab sells SAST as another way to do static analysis.

PMD out of the box: built-in rulesets

The easiest way to use PMD is in an IDE with its built-in rulesets. It has rulesets for languages from C++ to Scala, including Java. For Java there are a number of rulesets, which group related rules. Let's look at a few examples of rules.

- SimplifyConditional: [design ruleset] detect redundant null checks

```
1 class Foo {
2   void bar(Object x) {
3     if (x != null && x instanceof Bar) {
4       // just drop the "x != null" check
5     }
6   }
7 }
```

  Note that this code is not wrong. It's just redundant.
- UseCollectionIsEmpty: [design ruleset] better to use `c.isEmpty()` rather than `c.size() == 0`

```
1  class Foo {
2      void good() {
3          List foo = getList();
4          if (foo.isEmpty()) { /* ... */ }
5      }
6
7      void bad() {
8          List foo = getList();
9          if (foo.size() == 0) { /* ... */ }
10     }
11 }
```

  Again, it's not wrong to call `size()` and see if the result is 0. It's just more idiomatic, and sometimes more efficient, to check `isEmpty()`.
- MisplacedNullCheck: [basic ruleset] don't check nullness after relying on non-nullness

```
1 public class Foo {
2     void bar() { if (a.equals(baz) || a == null) {}     }
3 }
```

  The check `a == null` is never going to succeed, because `a.equals()` would throw a `NullPointerException` instead. So if `a` can ever be null, there is a fault.

[4] `pmd.github.io`

- UseNotifyAllInsteadOfNotify: [design ruleset] most of the time, `notifyAll()` is the right call to use, not `notify()`. Unless you know what you're doing, using `notify()` is going to result in a bunch of stuck threads, which is a bug.

```
1 class Notifier {
2     void bar() {
3       synchronized(this) {
4         this.notify(); // should likely be .notifyAll()
5       }
6     }
7 }
```

Find more about the above rules at `https://pmd.github.io/pmd/pmd_rules_java.html`.

I've included examples from the design and basic rulesets. There's a ruleset specifically for JUnit, rule sets detecting empty or otherwise useless code, naming conventions, and much more.

It is also possible for you to write your own rules. PMD supports XPath queries or Java visitors for user-supplied rules. SemGrep is a tool that supposedly makes it easier to write rules compared to PMD: the marketing copy says "Semgrep rules look like the code you already write".

Strengths. In addition to tabs vs spaces, these tools are especially good at identifying well-known security vulnerabilities that are present in your code—for instance, uses of unsafe APIs. (The easiest example of that is calling C's `gets()` function; use `fgets()` instead. The documentation for `gets()` is pretty clear: "Never use `gets()`.") It is possible to write custom checkers for new vulnerability classes as they come out.

Static analysis tools are exhaustive, so they are also good at catching things like missing error handling, or other edge cases.

Limitations. The flip side is that because static analysis tools are exhaustive, they may return false positives. This is especially the case for patterns that cross method boundaries. It's hard to reason interprocedurally, both for people and for computers. If you get too many false positives, you will ignore the false positive producer.

PMD and tools like it can tell you about things that may be wrong, or that are certainly wrong. However, they cannot tell you about how important that wrongness is. We still need (experienced!) human judgment to know that.

Writing your own PMD rules

Now we'll talk about how to write your own PMD rules. The intellectual core of a PMD rule is a query on the Abstract Syntax Tree (AST). You can use either Java or XPath to describe this query. XPath is cleaner, in that it's a declarative query language.

Here are some links about how to make rule sets and the boilerplate you need for rules:

- `https://pmd.github.io/latest/customizing/howtomakearuleset.html`

- https://pmd.github.io/latest/customizing/howtowritearule.html

We'll be focussing on what goes into the rule itself, as per https://pmd.github.io/latest/customizing/xpathruletutorial.html. The tutorial skips a lot of detail about how to actually use XPath. So let's start with that.

XPath. Let's start from the fundamentals. You write *selectors* to find nodes. We'll look at a simple XML document. XPath also applies to web programming (in particular the Document Object Model) and also to the Java code we'll be analyzing. Source: https://www.w3schools.com/xml/xpath_syntax.asp; specification: https://www.w3.org/TR/xpath/.

Here is an XML file:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <bookstore>
4   <book>
5    <title lang="fr">Harry Potter</title>
6    <price>29.99</price>
7   </book>
8   <book>
9    <title lang="en">Learning XML</title>
10   <price>39.95</price>
11  </book>
12 </bookstore>
```

You can observe the tree structure of the file. And you can play along with either https://www.w3schools.com/xml/tryit.asp?filename=try_xpath_select_cdnodes (which is hardcoded to XML similar to the above) or else http://www.freeformatter.com/xpath-tester.html.

Consider XPath expression `//price`. The result is the set of price nodes with data `29.99`, `39.95`. So, expression `//price` selects nodes with name `price`; the `//` means any descendants (including self) of the context node (= root node, here)—we asked for all descendants of the root named `price`. And, `count(//price)` counts the number of `price` elements in the tree.

We can also specify an exact path through the tree, say with `/bookstore/book[1]/title`. This starts at the root, visits the `bookstore` element, then its first `book` child, then returns the title. If we omitted `[1]`, then we'd get all of the titles.

We can also select all elements that satisfy some condition, e.g. `/bookstore/book[price>35]/title` selects the titles of books with price greater than 35.

Note the `lang` attribute. We can select elements with a certain value for `lang`: `//title[@lang="fr"]`.

In general, square brackets can contain predicates. We've seen pretty simple ones, but you can put arbitrary tree queries, e.g. `//price[../title[text()="Learning XML"]]`.

You can also combine predicates with `and`, `or`, etc. e.g. `//title[../price < 35 or @lang="en"]`. `*` works as you might expect.

The double-slash `//` includes descendants and self. If you want descendants excluding self, write

e.g. `descendant::book` as part of your expression. For the above example, there's no difference, but you can see it here:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
 <book><book><title lang="fr">Harry Potter</title></book></book>
 <book><title lang="en">Learning XML</title></book>
</bookstore>
```

Also, try `//book[descendant::book]` versus `//book[//book]`.

Chaos Monkey

Instead of thinking about bogus inputs, consider instead what happens in a distributed system when some instances (components) randomly fail (because of bogus inputs, or for other reasons). Ideally, the system would smoothly continue, perhaps with some graceful degradation until the instance can come back online. Since failures are inevitable, it's best that they occur when engineers are around to diagnose them and prevent unintended consequences of failures.

Netflix has implemented this in the form of the Chaos Monkey[5] and its relatives. The Chaos Monkey operates at instance level, while Chaos Gorilla disables an Availability Zone, and Chaos Kong knocks out an entire Amazon region. These tools, and others, form the Netflix Simian Army[6].

Jeff Atwood (co-founder of StackOverflow) writes about experiences with a Chaos Monkey-like system[7]. Why inflict such a system on yourself? "Sometimes you don't get a choice; the Chaos Monkey chooses you." In his words, software engineering benefits of the Chaos Monkey included:

- "Where we had one server performing an essential function, we switched to two."
- "If we didn't have a sensible fallback for something, we created one."
- "We removed dependencies all over the place, paring down to the absolute minimum we required to run."
- "We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available."

[5] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`
[6] `http://techblog.netflix.com/2011/07/netflix-simian-army.html`
[7] `http://blog.codinghorror.com/working-with-the-chaos-monkey/`