CITS5501 Software Testing and Quality Assurance
Semester 1, 2022
Workshop 7 (week 8) – Reviews – solutions

Reading

It is strongly suggested you complete the recommended readings for weeks 1-7 *before* attempting this lab/workshop.

A. 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 having someone proofread a written essay or assignment before you submit it.

It has two main 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, as well as in industry.

Most companies and large projects have coding style standards (for example, the 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. We won't get into that much detail, but we stress that

- it's important to be self-consistent
- when working on a team project, it's *very* important to follow the conventions or style guide adopted by that project.

If you're interested in getting more details about how code review works, then there is more information available in the *Pressman* textbook, as well as a whole StackExchange site dedicated to code review.

B. Exercise

If you are attending this workshop face to face, then for this exercise you should split up into small groups of 2-3 people.

If you are attending online via Teams, your facilitator should be able to create "breakout rooms" for you, by following the MS Teams instructions; but failing that, instead do a solo review of the code, and report back to the class as a whole with your findings.

For each of the following code samples, each person in a pair or group should take the time to read through the code carefully, bearing in mind the "Code review instructions" at the end of this sheet and recording problems they find (and possibly suggesting concrete improvements). Spend about 5 minutes on this. Then with your partner or group, discuss what problems you have found, and see if you agree or disagree on what they are (5–10 mins).

Then share this with the class as a whole; your facilitator may suggest some more.

Code sample 1 – `dayOfYear`

```
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;
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     } else if (month == 8) {
15         dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
16     } else if (month == 9) {
17         dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
18     } else if (month == 10) {
19         dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
20     } else if (month == 11) {
21         dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
22     } else if (month == 12) {
23         dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
24     }
25     return dayOfMonth;
26 }
```

What problems can you see with this code? What improvements would you suggest?

Does it violate any of the guidelines given in the "Code review instructions"?

Some problems with code sample 1:

- No Javadoc comment: there's no specification or explanation of what the method is supposed to do.
- Confusing order of arguments: they are in the order month, then day, then year. More logical would be day, month, year (or year, month, day). In languages with keyword arguments, like Python, you could do better still, and insist the caller clearly label each argument with what it is supposed to represent.
- Violates the DRY principle: the code is *extremely* repetitive. For instance, the number of days in January (31) is repeated many times through the code.
- Contains "magic numbers".
- The result is actually *wrong* for any year that is a leap year. In the improved code sample below, we assume we have access to an `isLeapYear(int year)` method.
- One purpose for each variable: the parameter `dayOfMonth` is re-used to store the result. It's better for each variable to serve one clear and distinct purpose.

There are many possible ways to improve this code, but here's one (quick) possibility:

```
/** Given a date, specified by providing a day of the month (bounds
 * depend on the exact month, but should always be between 1 and 31
 * inclusive), a month (from 1 to 12), and a 4-digit year,
 * return what day of the year that date occurs on.
 */
public static int dayOfYear(int dayOfMonth, int month, int year) {
  // For non-leap years: days in Jan-Nov.
  int monthLength = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31 };
  int result = dayOfMonth;

  // e.g. days in Feb have added to them the no. of days in Jan
  // (array pos 0) added to them, only.
  for int (i = 0; i < month - 1; i++) {
    result += monthLength[i];
  }
  // months past Feb are affected by leap years
  if (isLeapYear(year) && month > 2) {
    result += 1;
  }
  return result;
}
```

It too could still be improved:

- The Javadoc should explicitly document each parameter and the return value.
- We could add to the Javadoc an explanation of what happens if out-of-bounds values are supplied (e.g. -1 or 0 for the month) – probably, throwing an exception would be most appropriate.

(Note that in *this* context – improving a piece of code – it's perfectly fine to alter the specifications/design. But when writing *tests* for a method, the only thing you can test against is what the documentation says.)

- Rather than using magic numbers for months – we could define a Java enum:

```
enum Month {
  JAN = 1,
  FEB,
  MAR,
  // ... etc.

```

Then callers (and readers of the code) wouldn't have to guess at what `month > 2` means (it'd be written `month > FEB`).

Code sample 2 – `leap`

Hopefully you spotted one problem with the previous code sample – it gives the wrong answer, when the year is a leap year!

So suppose a colleague writes the following method, intended to determine whether a year is a leap year. Again, look for problems in and potential improvements to this code (5 mins), then discuss these with your partner(s) (5–10 mins).

```
1  public static boolean leap(int y) {
2      // convert to string so we can access 1st, 2nd digit etc
3      String tmp = String.valueOf(y);
4      if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 ||
5          tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
6            if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true; /*R1*/
7            else
8                return false; /*R2*/
9      }else{
10         if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
11             return false; /*R3*/
12         }
13         if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')
14           return true; /*R4*/
15     }
16     return false; /*R5*/
17 }
```

Some problems with code sample 2:

- Still no Javadoc comment!
- Also, there's no need for a one-letter parameter name: why not call the `y` paramater `year`?
- The year is converted into a String, simply so we can access particular digits. This is needlessly expensive (at worst), and an obscure way of doing things (at best).
- The method could be named better. Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases.
- Comments have been added, but they are cryptic – there's no information as to what "R1", "R2" etc are. (Though presumably they're some sort of reference to different bits for the algorithm for determining a leap year, https://en.wikipedia.org/wiki/Leap_year#Algorithm. Perhaps this one: https://docs.microsoft.com/en-us/office/troubleshoot/excel/determine-a-leap-year#how-to-determine-whether-a-year-is-a-leap-year.)

Don't do this! If trying to implementing a specific, publicly available algorithm or data structure, give a citation so people know exactly what you're using. e.g. "This implementation of a Red-Black Tree is based on Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 3rd edn, chapter 13."

- Code formatting is very inconsistent, making it harder to understand what the method is doing (e.g. why is the "return" statement in line 6 on the same line?)
- The method makes the assumption that the year is a 4-digit year (if it's only 2-digit, expressions like `tmp.charAt(3)` in line 13 will throw exceptions). This assumption should be clearly documented (or better yet, not made at all – what if historians or astronomers want to use this code for proleptic Gregorian years in the 1st century A.D.?)
- It uses an extremely obscure algorithm, and gets many years wrong, anyway – try 1600 or 1552.

There are many possible ways to improve this code, but here's one (quick) possibility, which makes improvements including the following:

- a good practice is to name methods that return booleans "is ... (something)" or "are ... (something)".

```
1  /** Returns whether the proleptic Gregorian calendar year
2    * <code>year</code> is a leap year.
3    *
4    * Uses the algorithm described in *Calendrical Calculations* by
5    * Reingold & Dershowitz, chapter 3.
6    */
7  public static boolean isLeapYear(int year) {
8    if (year % 4 != 0) {
9      return false;
10   } else if (year % 100 != 0) {
11     return true;
12   } else if (year % 400 != 0) {
13     return false;
14   } else {
15     return true;
16   }
17 }
```

If following a well-known algorithm (especially one this short), inline comments probably aren't necessary (but Javadoc documentation still is).

C. Linters and formatters

In general, it's expensive to spend the time of humans doing something a computer could do equally well – like formatting code correctly, or spotting errors that can be checked for mechanically. So before a human reviews your code, you should ensure it's already been checked for basic errors by a computer.

Programs called *code formatters*, *code beautifiers* or *pretty-printers* apply consistent formatting to code. Some apply a single rigid, built-in style (e.g. "all indents are 4 spaces") but most can be customised to reproduce a style the team prefers. One commonly-used formatter used in Java projects is *Spotless*, which can be integrated easily with the Microsoft Visual Studio Code editor, or the IntelliJ IDEA IDE. If you use either of those tools to write your Java code, you might like to experiment with it.

Linters analyse your source code looking for common programming errors, poor style, and programming language constructs which should be used with caution. They are performing a simple form of *static code analysis* (which we'll talk more about later). Examples of linters for Java include Checkstyle, Spotbugs, and PMD. In your own time, you might like to try running one of these tools over code you have written, to see what sort of problems they can flag.

D. Challenge exercise

You don't need to complete this exercise in class, but if you have a good familiarity with coding, you might like to attempt it as an extra challenge (either in class, if there is time, or in your own time).

Suppose you are desiging a custom database system, PriceWise, intended to allow managers of retail stores to query catalogs and stock.

Amongst the things it must do are the following:

- Provide a GUI, in which users can enter a query in a custom query language, RQL (Retail Query Language), specifically written for retail systems
  - This will involve parsing the RTL query and using it to consult an internal database
- Interact with point-of-sale (POS) systems and warehousing systems to determine when stock has been added or sold.

1. Try and sketch out as a system block diagram what you think some of the major components or subsystems of the PriceWise system might be.
2. Of the different testing techniques we have seen in the unit so far, which might you apply, and to what parts of the system?
3. What would you development/integration testing approach be? Would you develop and test the system top-down, bottom-up, or in some other way?

E. 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 in this workshop we looked at code samples which highlightede the following general principles:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Avoid magic numbers
- One purpose for each variable
- Use good names
- Use whitespace for readability

These general principles help ensure that code is:

- **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.
- **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.

Appendix – Code review instructions

This document describes how and why to perform code reviews.

You can't learn how to write without learning how to read – and programming is no exception. Code reviewing is widely used in software development, both in industry and in open source projects. Some companies like Google even make it a policy that *all* code must be reviewed before being committed to version control – this means that for every single line of code in Google's repository, another software developer has read it, given feedback on it, and signed off on it.

Code review can be seen as a type of *static quality assurance* technique (it doesn't require the code to be run), and has multiple benefits:

- It helps find bugs, in a way that's complementary to other techniques (like testing and static analysis)
- Reviewing uncovers code that is confusing, poorly documented, unsafe, or otherwise not ready for maintenance or future change.
- Reviewing spreads knowledge through an organization, allowing developers to learn from each other by explicit feedback and by example.

So code reviewing is not only a practically important skill that you will need in the real world, but also a learning opportunity.

What to look for

In the exercises for this workshop, you can make comments about *anything* you think is relevant. But some guiding principles are that we want code to be:

- free from bugs
- easy to understand, and
- easy for another developer to change.

So read the code with those principles in mind. What follows are some concrete examples of problems to look for – feel free to add others.

Bugs or potential bugs.

- Repetitive code (remember the "DRY" principle – "Don't Repeat Yourself").
- Disagreement between code and specification.
- Off-by-one errors.
- Using global variables when a smaller scope could be used.
- Optimistic, undefensive programming.
- Magic numbers.

Unclear, messy code.

- Bad variable or method names.
- Inconsistent indentation.

- Convoluted control flow (if and while statements) that could be simplified.
- Packing too much into one line of code, or too much into one method.
- Failing to comment obscure code.
- Having too many trivial comments that are simply redundant with the code.
- Variables used for more than one purpose.

Misusing (or failing to use) essential design concepts.

- Incomplete or incorrect specification for a method or class.
- Exposing (as `public` or `protected`) implementation details that should be kept `private`.
- Invariants that aren't really invariant, or aren't even stated.

Positive comments are also a good thing. Don't be afraid to make comments about things you really like, for example:

- unusually elegant code
- creative solutions, or
- great design.

Credits

The code samples, instructions and exercises in sections A–B and E are adapted from MIT's course 6.005, "Software Construction", reading 4, "Code Review", available from the MIT website, which was collaboratively authored with contributiopn by Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. This worksheet (as well as the original MIT content) is licensed under the CC BY-SA 4.0 license.