Project Two

Brian Kittrell

Southern New Hampshire University

CS 320: Software Development Life Cycle

Dr. Albanie Bolton

October 15, 2022

In testing my project, I adhered to the guidelines laid out by the software's requirements throughout in order to meet all of the testing and design goals. First, I was thorough in my testing and made sure that each class, method, and parameter had a test that touched it directly or indirectly. Second, my tests were well-organized and written in a way that it was considerably easier to design tests for the program's classes as we proceeded through the course, which showed me that the tests were complete and performing correctly. Additionally, I reached nearly 87% testing coverage in my code, I was exceeding the rubric requirements by a significant margin, and many teams in real world scenarios may not meet even an 80% coverage percentage in testing (Heusser, 2021). The high coverage percentage, zero error rate, and the completeness of my tests told me that the code was solid enough for production.

Technical excellence is a personal goal of mine, and I ensured that my code was technically sound by making sure it functioned efficiently, performed its intended tasks, and tested accurately. It went further than the project requirements in a few areas to match more secure, real-world practices and eliminated edge cases often resulting from human error. For instance, my ID generation code is shown here.

```
  private static String setId(int idLength) {
    String alphaNumericString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                + "0123456789"
                + "abcdefghijklmnopqrstuvxyz";

    StringBuilder generatedId = new StringBuilder(idLength);

    for (int i = 0; i < idLength; i++) {
          int index = (int)(alphaNumericString.length() * 
Math.random());
          generatedId.append(alphaNumericString.charAt(index));
    }
    return generatedId.toString();
    }
}
```

This segment takes the idLength limit from earlier in the class to generate an ID string of alphanumeric characters in the spirit of a UUID, build the string, and return it to the constructor when a new record is instantiated. Since it is only called in the constructor, the method and resulting property are designated with the private accessor keyword, the ID is stored as an immutable value. It was impossible to test on its own in JUnit for this reason, but performing simple feedback testing during the design of the method yielded sufficient results. Further testing of dependent methods which relied on this generator proved successful, which vicariously proved that the generator was sound. This mirrors real-world applications that generate an ID that must be unique, must maintain data integrity by being permanently fixed, and has a very low risk of collision.

Further, I maintained a technically sound design pattern in using getters and setters for all values in the class and resulting objects so that validation and manipulation–if desired in the future–could be performed on these values without refactoring large areas of code. Along with that, rules are assigned to variables whenever possible to maintain freedom of customization and high maintainability. The software is truly object-oriented.

Efficiency should always be in mind when designing any application, and it is best practice to avoid long logic chains and complicated loops, not only for the final efficiency of the program but for the efficiency of designing and implementing the features of the program. There was a time that I designed and coded for every possible edge case that could arise in the program, but I recognize that it was short-sighted and taking on too much on the design side.

Users should be allowed–and expected–to think their way through some issues when encountered.

One example of this principle is in the following code:

```
public static boolean validate(String input, String type) {
        // private byte nameLength = 20, descLength = 50; // Input
length rules
     boolean result = false;
        if (type == "name" && input.length() > 0) {
            if (input.length() > nameLength || input == null ||
input == "") {
                System.out.println("Name length cannot be greater
than 20 or empty on name: " + input);
                result = false;
            } else {
                result = true;
            }
        }
        if (type == "description" && input.length() > 0) {
            if (input.length() > descLength || input == null ||
input == "") {
                System.out.println("Address length cannot be
greater than 50 or empty.");
                result = false;
            } else {
                result = true;
            }
        }
        return result;
    }
```

In the validate method example, rules are tested for broad adherence on a "catch all" basis, ignoring potential underlying edge cases that make exist because they are either inclusive of the broader rule or are too specific or contrary in nature.

The two main testing techniques I employed were assertions and exception handling tests. These two yielded the best results. Assertion tests take in a boolean argument and a method or constructor from the class and determine if the return value matches the boolean given. Exception handling tests involve feeding correct or erroneous arguments to a method or constructor to see if the software produces an exception of the correct type or including a specific error message. Both of these imply that the software is or is not giving the expected result per the engineer's design and reflect areas which may need to be changed to produce the intended outcome in projects.

I did not employ manual testing much in this project like I usually do, but the software is currently in a state without a user interface. As a result, testing had to be undertaken directly on the classes, constructors, and methods in code, which is something with which I was not terribly familiar, but in the end, I enjoyed learning about the process and can see clear application for this kind of testing in the future. Obviously, the merits of manual testing cannot be overstated; software that does not undergo manual testing at some stage–preferably every stage–cannot be considered vetted for production.

It was important to be cautious of the complexity of the code and perform targeted tests in simple ways due to how the code was interdependent much of the time. However, the code was designed in a modular fashion, which made testing individual parts on their own merit easier to handle. I wasn't required to call multiple blocks in strange ways from the test code in order to isolate specific parts for results, which is one of the strengths of object-oriented programming and why we use it.

When I write code, I can be hard on myself, so my bias rarely comes from a place of pride until I've attempted to break the code in every conceivable way. Even so, I recognize that testers will likely find some problem or another, and having thick skin and a good sense of humor have helped in that regard in the past. Bias is one of our greatest enemies when creating in general regardless of the resulting product. Those who cannot take criticism in their work cannot grow, although offering criticism can be done in ways so as not to insult or degrade the creator.

It is critical to my success as a software engineer to be disciplined in my commitment to quality and completeness in testing. In some cases, I employed multiple tests to individually test each of the possible arguments of each method for accuracy, even when I knew that the same method was handling the validation, and I did this because I may not always be the developer and the tester on the same product. On the same note, overtesting can result in technical debt just as surely as badly written code that needs to be refactored, so there are limits. It is less risky and less likely compared to rushing through a project, but it can be an additional time waster.

In conclusion, the exercises throughout the course have been invaluable to me, and I believe that the usefulness of every lesson learned here will find application throughout my career going forward. The projects were challenging and complex, especially for someone who doesn’t work in Java much of the time, but they strengthened my core skills and taught me many new things along the way. At first, I was apprehensive about JUnit, but once I was in the swing of things and had some experience writing test cases, the work became much smoother. All in all, I am thankful to have had the experience of this class, and I feel it will benefit me for a long time to come.

References

Heusser, M. (2021, December 15). *What unit test coverage percentage should teams aim for?* SearchSoftwareQuality. Retrieved October 16, 2022, from https://www.techtarget.com/searchsoftwarequality/tip/What-unit-test-coverage-percentage-should-teams-aim-for