Project Two

Anthony Lee

SNHU

CS-320-T4512 – Software Test Automation& QA

Professor Alex Pettit

April 12, 2022

**Summary**

My unit testing approach was to test each element of the software where input is required. For example, if the input’s requirement was to add a task, then my tests would first testing if the adding function worked as intended. That would include testing if the add function would add a task and testing if the add function would not add a task that did not meet the requirements of a task. What this looked like is with the task requirements being a unique task ID with a string no longer than 10 characters:

**public** **void** testAdd() {

// Create task list

TaskService cs = **new** TaskService();

// New tasks to add

Task test1 = **new** Task("1", "name", "description");

// If the information is passes, add to list

*assertEquals*(**true**, cs.addTask(test1));

@Test

**public** **void** testAddFail1() {

// Create task list

TaskService cs = **new** TaskService();

// New tasks to add

Task test1 = **new** Task("1", "name", "description");

// If the information is passes, add to list

*assertTrue*(cs.addTask(test1));

// Try to add information that is already in the list

*assertEquals*(**false**, cs.addTask(test1));

The add function will fail in the failure test because the task ID “1” already exists in the data structure. The point of all of this was to not only test if the function would work, but if the function would work as intended. This required making a minimum of two tests; one test to run correctly and at least one test to make the function not run with inputs that do not meet the requirements. The overall quality of my JUnit tests is strong with a coverage percentage of at least 90%. The reasoning behind the coverage being at 90% is due to using if statements in the conditions, which can lead to a lower coverage percentage.

My experience writing JUnit tests shows they were technically sound and efficient by both the coverage percentage and the simplicity. Each test was written in a way that only one aspect of the requirements was tested at a time, reducing the number of objects tested in each JUnit test. This way the focus of the test would be the only focal point for a test passing/failing. For example, the requirements for a task ID are that it is unique, no longer than 10 characters, and is not null and not updatable. The uniqueness of the task ID was illustrated above, and the tests for the other requirements are:

// testing task id length

@Test

**void** testTaskIDTooLong() {

// generate string with length of 11

**int** width = 11;

**char** fill = '0';

// padding the string

String toPad = "1";

String idTest = **new** String(**new** **char**[width - toPad.length()]).replace('\0', fill) + toPad;

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Task(idTest, "name", "description");

}); }

// testing task id for null

@Test

**void** testTaskIDIsNull() {

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Task(**null**, "name", "description");

}); }

Note that the requirement for being non updatable was taken care off with the string for task ID being final in declaration in the constructor. I tested both requirements separately because if I had tested both at the same time, then the failure test would throw an illegal argument exception at the first requirement failure and would not run the rest of the test. While I could write the test to not do that, separating the requirements also allows others who look at the JUnit test code to see which requirement each test is focusing on. Although more tests were created, having a test to focus on only one requirement at a time provided a technically sound test. Also, having each test only test one requirement made each test shorted in lines of code written, thus improving the efficiency of each test.

**Reflection**

The software testing technique I employed for each milestone is white box testing because I was testing knowing what the internal structure of the code is. I chose inputs to explore paths and determine appropriate output. This allowed me to test individual component of the code to verify functionality (Chacon, 2018). I used JUnit testing to do white box testing. For example, when I was testing the component for adding a new object to the data structure, I first tested if the function would work by giving it correct information that passes all the conditions for a correct input. Then I tested the adding function by forcing it to fail by giving it information that fails conditions to add it to the structure. Doing JUnit testing is a way I did regression testing, which is to see if changing the code broke existing functionality.

Other software testing techniques I did not use for the milestones is black box testing. In black box testing, testers test the software without knowing the internal structure of code or program (Johnson, 2020). Because I wrote the code, I knew the internal structure of the code. Black box testing involves testing the entire system instead of individual components; I tested each individual component for each milestone.

White box and black box testing examine and evaluate the code base in different ways. According to Johnson (2020), white box testing examines the software’s internal structure, thereby allowing testers to access the source code, syntax, design, architecture, and configuration to see if the code follows the design. I used white box testing to test if the milestones’ requirements were met. Using white box testing allows software developers to test their code base while working on their projects. On the flip side, black box testing is beneficial for testing software usability, giving a broader picture of the software from a user’s perspective (ReQtest, 2019). The practical use for doing black box testing is to see how the system works as a whole instead of individual components that white box testing does.

The mindset I adopted while working on this project was to be as objective as possible. That meant the tests I created needed to focus solely on the requirements and nothing more. Even though my coverage was not at 100% from my JUnit tests, chasing down the last ~10% would lead to me creating more exceptions in the code which could create more errors down the road because with my current knowledge of this code I would not be able to get a 100% coverage. By only focusing on the requirements of the project, I was able to accept that the complexity and interrelationships of the code would result in a less than 100% coverage.

When reviewing my code, limiting bias is important. On the software developer side, having bias towards your code could result in shipping software that only work in specific conditions therefore delivering a poor product. The way I tried to limit my bias was to focus only the requirements and only test one requirement at a time. As previously illustrated above, by testing one requirement at a time, I was able to make sure that each requirement was met.

The key to being disciplined in my commitment to quality is to be as thorough as possible. The importance of not cutting corners when it comes to writing and testing code is to be able to deliver a product that both meets clients’ requirements and be as technically sound as possible. To avoid technical debt, I made sure that each test in this project was commented to let others know what each line of codes purpose was, and each function and test were given names that explained what they were for. As shown above, each test had been named as test followed by what was being tested. Also, each test had comments stating what the following lines were for.

References

Chacon, D. (2018, August 2). *JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests.* Parasoft.com. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

Johnson, P. (2020, September 24). *Black Box Testing: What You Need to Know.* whitesorcesoftware.com. https://www.whitesourcesoftware.com/resources/blog/black-box-testing/

ReQtest. (2019, August 8). *Black Box Testing – Understanding the Basics.* Reqtest.com. https://reqtest.com/testing-blog/black-box-testing/#:~:text=Unlike%20traditional%20white%20box%20testing,beneficial%20for%20testing%20software%20usability.&text=Black%20box%20testing%20gives%20you,system%20rather%20than%20different%20modules.