**7-2 Project 2**

In Project 1, each of the software requirements was translated into a unit test. For example, in Task.java, the task name was analyzed before being accepted. According to the requirements, the task name was checked to see if it was null or empty and to ensure it was fewer than 20 characters long. Similar checking was performed for the task description: the description may not be null or empty, and it cannot be longer than 50 characters. The coverage percentages of the JUnit tests give a good sense of the percentage of the code that is being tested. Also helpful are the red, yellow, and green highlights which expose the sections of the code being tested and the results of the testing. It was satisfying to confirm that newly written tests indeed covered more of the code by observing the newly highlighted green code and seeing the JUnit percentage coverage increase.

To ensure the code was technically sound, it was necessary to consider all possible inputs the system could receive, and how the system should respond to each possible input. For example, if the name of a task is empty, we want to throw an error, but we don’t want to interrupt the entire program. Hence, to keep the code technically sound, the null and empty cases of input result in a task with the title “NULL”. This way, the program can continue running while having flagged the task for its faulty name. This error is handled in the following JUnit test in TaskTest.java:

*@Test*

*@DisplayName*("Name cannot be null")

void testNameNotNull() {

Task task = new Task(null, "Description");

if (task.getTaskName() == "NULL") {

*assertNotNull*(task.getTaskName(), "Name was null");

}

}

if (name == null || name.isEmpty()) {

this.name = "NULL"; // null or empty

}

The name input was null but thanks to the input sanitization in Task.java, the name does exist and it’s simply “NULL”. Combining input sanitization with JUnit tests for all branches of if-else statements allows us to anticipate and identify bugs quickly. This is an example of how we ensure the code is technically sound in the case of faulty input.

JUnit tests also help the code run more efficiently. Each requirement comprises its own JUnit test, and it is easy to tell whether or not a test has failed or succeeded. If a test needs rewriting, the resulting area is highlighted in red. This expedites the refactoring of code. Debugging would take much longer if the code was not divided into JUnit tests. These tests simplify the process of locating bugs in the code and provide guidance on what needs to be changed.

In this project, I used the software testing techniques of manual testing, functional testing, unit testing, and regression testing. Integration testing, testing how the blocks of code interact with each other, could also be tested by including a main.py file which calls each method. Without the driver Python file, we are assuming that the collection of passed unit tests will work together smoothly. The testing was manual in that the tests were written from scratch and were called as part of the JUnit testing suite. No overarching testing framework like Selenium was used to write or call tests, which would be automated testing. We performed manual testing. For example, we wrote a test for each possible input of names and descriptions, until each if-else statement had all of its branches tested. Also, the testing performed was functional testing, which is similar to acceptance testing. The difference is that functional testing ensures that the code requirements are being met, while acceptance testing reflects if the user is satisfied with how the entire project turned out. Naturally, acceptance testing will include functional testing, but it will also involve testing multiple how features interface with each other, like in integration testing.

In testing the code, it was useful to have the mindset of a user (or hacker) trying to break the system. Is it possible to crash the code by using too many characters? Will it ruin the design of the cards to have too many characters in the description? While testing the code, it was important to develop “CRUD” functionality: create, read, update, and delete. Testing these methods independently was important because if things worked well on a unit-test level, then they are set up for success to work well on a system-level, as could be confirmed through integration testing with a driver Python file.

If I were a developer I would have a bias about my work. I would write fewer JUnit tests, since I would have confidence in my abilities that not all tests were necessary. For example, after the creation of a new Contact, the getters for each component of the contact are tested: getFirstName(), getLastName(), getAddress(), and getPhoneNumber(). As a software engineer, I may assume these methods work properly because there is little room for error. I would also feel pressured to continue to the next assignment to be productive. It’d be helpful to have a dedicated QA engineer on the team whose job to search for and test vulnerabilities in the code, whether the bugs are obvious or subtle. All tests can be compiled into a test file, and the coverage and success percentages can be easily checked. If any test turns out to be a failure, the QA engineer can contact the author of the faulty code and help them refactor it until it passes the test(s).

It may be tempting to cut corners in software development by making code that is “good enough”, say, good enough to work 95% of the time. Also, it may be tempting to write tests that cover “most of” the code, say, 60%, and assume that if 60% of the code passes all the tests, then the remaining 40% will pass as well. However, these sample sizes are not always sufficient to catch bugs that would otherwise make it into the finished product.

Writing code with these relaxed standards also creates technical debt. Technical debt is error made during coding sprints which prioritize speed of development over testing, documentation, and best coding practices. Best coding practices include making clear comments for future users and consistent syntax choices. Since development practices which prioritize speed involve rapidly changing code, it wouldn’t be productive to test the code as it is developed. Instead, technical debt can be avoided by having a QA engineer use the software development time to create a suite of tests covering all the customer requirements. Once the development phase is finished, we can apply the tests and make final edits.