Module 7 Project 2: Summary and Reflections Report

Parker G McKay

Southern New Hampshire University

CS-3220-14822-M01: Software Test, Automation QA

Omar Toledo Lopez

April 20, 2025

Summary and Reflections Report

Summary:

For Project One, my unit testing approach centered around testing each operation's success and its different failure requirements. I made sure to cover not only the expected functionality but also edge cases and invalid input scenarios. Each feature—adding a task, updating task fields, and deleting tasks—was tested with the goal of ensuring that every technical requirement outlined for the Task and Task Service classes was properly verified through JUnit tests.

My approach was well aligned to the software requirements. For example, one requirement stated that the task ID must be unique, cannot be null, must not be updatable, and cannot exceed 10 characters. To meet this, I wrote a test ensuring that creating a task with a null ID or an ID longer than 10 characters would throw an exception. Specific evidence includes my test method:

@Test

void testTaskIdTooLong() {

assertThrows(IllegalArgumentException.class, () -> new Task("12345678901", "Name", "Description"));

}

Similarly, for the name and description fields, which were limited to 20 and 50 characters respectively and could not be null, I created tests such as:

@Test

void testTaskNameNull() {

assertThrows(IllegalArgumentException.class, () -> new Task("12345", null, "Description"));

}

@Test

void testTaskDescriptionTooLong() {

assertThrows(IllegalArgumentException.class, () -> new Task("12345", "Name", "This description is way too long and exceeds the fifty character limit set in requirements."));

}

For the Task Service class, the requirements demanded functionality to add, delete, and update tasks based on the task ID. My unit tests focused on verifying these operations while handling edge cases such as attempting to delete a nonexistent task or updating a task with invalid input. For example, to ensure that deleting a task only worked when the ID existed, I wrote:

@Test

void testDeleteNonexistentTask() {

TaskService service = new TaskService();

assertFalse(service.deleteTask("nonexistentID"));

}

I defended the overall quality of my JUnit tests by aiming for thorough coverage of both success and failure scenarios. Based on the coverage report, I achieved over 90% code coverage, which gave me confidence that the majority of the logical paths in the application were exercised. Each public method had at least one or two corresponding tests verifying both typical behavior and error handling.

My experience writing JUnit tests was initially complicated. When I first approached writing unit tests, I struggled with understanding how to structure assertions and how to simulate edge cases properly. However, after conducting additional research and getting guidance from peers and tutorials, I started to find writing JUnit tests enjoyable. It became rewarding to predict potential failures and confirm the system behaved reliably.

To ensure that my code was technically sound, I made sure to include specific and meaningful assertions in every test. For example:

assertNotNull(task.getId());

This line from my testTaskIDNotNull() method verified that every task created had a valid, non-null unique ID, as required by the project specifications. I also incorporated assertions to check that exceptions were thrown when invalid inputs were provided, ensuring that error handling was properly implemented across different methods.

In terms of efficiency, I used teardown methods annotated with @AfterEach to ensure that the state was properly reset after every test case finished executing. For example:

@AfterEach

void tearDown() {

service = null;

}

This approach helped prevent any unintended interactions between tests by cleaning up objects after each run. By resetting the TaskService instance to null after every test, I made sure that each test started fresh and independent, keeping the test suite efficient, reliable, and easier to debug.

Reflections:

I used **Unit Testing** to focus on testing individual units or components of the software. Specifically, I implemented **JUnits**, a popular Java testing framework. Unit testing ensures that each component functions correctly in isolation. By testing small, specific pieces of code (such as methods or classes), it allows for quick identification of issues and bugs in the early stages of development. This approach is particularly useful for ensuring that the underlying logic of each module behaves as expected. For example, testing a method that calculates a sum would ensure that the logic is working correctly, even before integrating it into larger systems.

Another testing techniques is **Manual Testing** and it involves human testers inspecting and testing the software manually. While this technique is useful for evaluating the user interface, accessibility, and other subjective aspects, I did not employ it in this project. Manual testing requires human intervention to interact with the software, making it more flexible for exploratory or usability-focused tasks. However, for the scope of this project, I chose to focus on automated testing (Unit Testing) as it provides a more structured and repeatable approach for the components being tested.

**Unit Testing** is crucial for ensuring the correctness of individual components before they are integrated into the larger system. It is particularly effective in projects that require high reliability and complex logic, providing a structured and repeatable approach for frequent testing. This technique helps identify issues early, especially in larger projects, and is essential in preventing regressions when new code is added. On the other hand, **Manual Testing**, although not used in this project, is invaluable for evaluating aspects like the user interface and accessibility, which automated tests may miss. It is most beneficial for exploratory testing, user acceptance testing (UAT), or scenarios where human judgment is needed. While more time-consuming, manual testing ensures a refined user experience, making it a necessary technique in projects focused on usability and design.

In working on this project as a software tester, I focused on caution, thoroughness, and objectivity. I understood the complexity and interrelationships of the code, where small changes could trigger larger issues. For instance, when testing the addUniqueTask() method in the TaskService class, I carefully checked the task list after each addition to ensure tasks were being added correctly and that no unintended side effects occurred. I tested scenarios such as adding a unique task and verifying its properties to ensure each part of the code worked as expected in isolation.

To limit bias, I approached the code objectively and avoided assuming correctness, especially when testing methods like updateTasks. In this case, when testing an invalid task update with an ID that didn't exist, I ensured that the original task remained unchanged, highlighting the importance of independent testing to prevent overlooking potential issues. If I were testing my own code, it would be easy to overlook such edge cases or assume that the code was perfect. A fresh perspective helps ensure that nothing is missed.

Finally, maintaining discipline in my approach to quality was key. Rushing through tests or skipping edge cases like the testBadUpdateTasks() method could result in technical debt, where small issues accumulate and cause bigger problems later. By focusing on rigorous testing, I aim to avoid this, ensuring that the code remains maintainable and reliable.