# **7-2 Project Two: Summary and Reflections Report**

Mark Baligad

Southern New Hampshire University

CS-320 Software Test, Automation QA

Professor Nicholas Shepard

10/20/2024

**Summary**

**Unit Testing Approach**

I was tasked with developing and testing the back-end services for an application that manages appointment, contact, and task services. My approach to testing utilized JUnit tests to ensure that each method performed as expected. The JUnit tests function as black-box tests that execute specific portions of the code and check whether the code returned the expected result (Hambling et al., 2019, p. 99). To ensure the testing aligned with software requirements, I focused on testing the expected results defined in the requirements by creating JUnit tests to verify the input validation and class behavior.

Beginning with the Task class, the software requirements specified that a Task has a non-null ID and that the ID cannot be longer than ten characters. In the Task code, I covered the creation of a task and input validation that would trigger an illegal argument exception if any of the requirements were not met. To verify that the Task class functions as required, I created a TaskTest class, including JUnit tests for both positive and negative tests. For example, I tested if a task can be created with valid values with a “testTaskCreationSuccess” test. Additionally, I tested if the illegal argument exception is thrown when the ID is null or longer than ten digits with my “testTaskIdIsNull” and “testTaskIdTooLong” tests.

To ensure complete test coverage, I ran my JUnit test suite using the “with coverage” option to generate a test coverage report. The report checks every class, method, and line of code and reports if a test case was created to test the behavior for that section. By reviewing the coverage report, I identified the untested lines of code and added new unit tests to cover those lines until I reached 100% line coverage. In addition to reviewing the requirements and confirming that each requirement had a test case to verify that behavior, I effectively tested the entire application.

**Experience Writing JUnit Tests**

I adhered to best practices while writing the code for each class and JUnit test suite to ensure my code was technically sound. Technically sound code is free of errors and bugs, so I executed my code to ensure it did not encounter any runtime errors. My code successfully compiled and executed all JUnit tests with no unexpected results. Secondly, I utilized JUnit tests to ensure the code performed its intended task and did not violate input validation rules. For example, my “testAppointmentDateInThePast” JUnit test ensures that a user cannot enter a date in the past. Finally, I ensured the code was robust and readable using descriptive method names such as “getTaskId” and “testAppointmentIdTooLong.” Any developer can review my code and understand the processes as my method descriptions and test cases function as the documentation for my code.

To ensure that my code was efficient, I ensured each method performed only a single task and did not needlessly repeat the same functions in different methods. An example of testing efficiency is in my unit tests. My “ContactServiceTest” class uses a “@BeforeEach” method to create a contact service for the following tests. Each subsequent contact service unit test checks a specific method, parameter, and result without overlapping with other unit tests. For example, I test the “updateContact” method multiple times with my “testUpdateContactFirstName,” “testUpdateContactLastName,” and “testUpdateContactPhoneNumber” unit tests, but each unit test checks that a different parameter is updated correctly. Following efficient testing techniques throughout my code helps with readability and reduces the number of times duplicate code is executed, which optimizes the efficiency of the application and testing process.

**Reflection**

**Testing Techniques**

I primarily used block box testing in the form of JUnit tests to test the backend services. Black box testing means the inputs are supplied to the application, and the output is checked against the expected result. The inner workings of the application code are unknown to the test. Additionally, I completed white box testing by debugging the code and inspecting it for errors. During white box testing, I statically analyze the code by reviewing it before it is run, and then I debug the code as it executes to ensure the logic is executed. Combining black box testing with white box testing helped to increase test coverage, catch defects, and verify that the application functions as expected.

Due to the limited scope of testing and deployment of this application, there were other forms of testing that I did not use. An example of an additional test technique is integration testing, which could have tested interactions between the different services, such as between the appointment and task services, to confirm that they interact expectedly. I did not use system testing, which would help to ensure system stability and behavior under use. Finally, I did not use user acceptance testing (UAT) to involve real users in testing the software. Once the application has a front-end interface, I could create real-world scenarios for users to test to complete UAT testing and ensure the application meets their expectations.

In different software projects, block box testing helps test a service's external behavior. White box testing is essential for optimizing and correcting the application code. Integration testing is helpful for large-scale applications with separate interacting components to verify their correct interactions. Systems testing ensures that the platform executing the code is stable and can handle the demand expected under a live load. Finally, user acceptance testing ensures that the application meets the expectations of users who will use it in a live environment.

**Testing Mindset**

After discovering that I missed creating a unit test to check for null values in my task service, I learned that achieving full test coverage in your test report is possible without thoroughly testing all requirements. I knew that in the future, I would need to be more cautious in my testing approach. I found it helpful to add comments listing the application requirements to both the classes and the tests so that I do not miss testing a critical requirement. Following this mistake, I updated my unit tests to cover the requirements thoroughly before completing the code project. For example, I created a “testAppointmentDateInThePast” test to ensure that I covered the requirement that an appointment date could not be created in the past, which is a critical test to prevent errors from escaping in later stages of development.

To limit bias in my tests, I kept my assumptions from developing the methods from affecting my tests. The primary concern for this project is the potential bias when developers test their own code because they may have the inherent belief that the code should work as intended (Hambling et al., 2019, p. 36). Even though I wrote the “addAppointment” method to prevent the addition of duplicate appointment IDs, I still tested this assumption with the “testAddDuplicateAppointment” unit test. This helped to restrict my bias as the developer by ensuring the code was thoroughly tested for any mistakes I might have made.

Finally, maintaining discipline and commitment to quality as a software engineer was critical in this project. Cutting corners during development or testing can lead to failures when the application is used in a live environment. Real-world examples of software testing failures, such as the Therac-25 catastrophe, prove that insufficient testing can lead to a loss of money, reputational damage, and even injuries or death (Buckleton et al., 2021). In the future, I plan to avoid technical debt and maintain quality by adhering to best practices, such as writing clear, maintainable code and addressing issues as soon as they arise rather than postponing fixes. An example of the quality in my code is how I moved my input validation to my setter methods and then reused these in the constructor to prevent repetition and make my code easier to maintain.

**References**

Buckleton, J. S., Curran, J., Taylor, D., & Bright, J. (2021). What can forensic probabilistic genotyping software developers learn from significant non‐forensic software failures? Wiley Interdisciplinary Reviews: Forensic Science, 3(2), 1–8. <https://doi.org/10.1002/wfs2.1398>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing: An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.