**Summary and Reflections Report: Project Two**

Michael T. Cary

Southern New Hampshire University

CS 320 Software Testing, Automation, and Quality Assurance

Jeff Phillips

October 19, 2024

Throughout the project, I implemented unit tests using JUnit for the three features: Contact Service, Task Service, and Appointment Service. My testing approach involved testing each class and method to ensure they adhered to the client’s requirements. Using black-box testing, I focused on the inputs and outputs, ensuring the correct behavior for valid and invalid cases without looking into the internal structure of the code.

For the Contact Service, unit tests verify the creation, updating, and deletion of contacts. Each test ensured that fields like contact ID, first name, last name, phone number, and address followed the set constraints. These constraints included things such as being non-null or limited to a certain character length. The same principles were applied to the Task Service and Appointment Service to ensure their fields adhered to the specified limits

The unit tests were aligned with the software requirements provided by the client. For example, the Contact Service requirement to have a (contactId) that cannot exceed 10 characters was tested using cases that included both valid and invalid inputs:

**Exception exception = assertThrows(IllegalArgumentException.class, () -> {**

**new Contact(null, "John", "Doe", "1234567890", "123 Street Name");**

**});**

**assertEquals("Invalid contact ID", exception.getMessage());**

This test case demonstrates how we enforced the constraints for the (contactId), ensuring alignment with the project’s requirements

The quality of my JUnit tests was validated through the coverage analysis tool in Eclipse, which showed 100% test coverage for all core features. Each feature’s functionality, including edge cases, was thoroughly tested. For example, handling edge cases for the Task Service involved testing for name length (20 characters) and description length (50 characters), ensuring that the service rejected any inputs that exceeded the defined boundaries.

Writing JUnit tests for this project was a valuable learning experience, especially when handling exceptions. One challenge I faced was ensuring all edge cases were covered. An example was when creating an appointment with a date in the past. This test required attention to detail:

**Exception exception = assertThrows(IllegalArgumentException.class, () -> {**

**new Appointment("123", new Date(System.currentTimeMillis() - 86400000), "Doctor's Appointment");**

**});**

**assertEquals("Invalid appointment date", exception.getMessage());**

This helped me understand the importance of catching all possible errors during testing.

To ensure my code was technically sound, I relied on strict enforcement of constraints like null checks and length validations. For instance, in the Appointment Service, I ensured that appointment dates couldn’t be set in the past:

**if (appointmentDate.before(new Date())) {**

**throw new IllegalArgumentException("Invalid appointment date");**

**}**

I then verified this with a corresponding test to ensure robustness in handling invalid dates.

I ensured the efficiency of my code by grouping related test cases, avoiding redundancy, and improving the clarity of the tests. For instance, instead of creating multiple tests for every invalid case, I grouped them in a single test method:

**exception = assertThrows(IllegalArgumentException.class, () -> {**

**task.setName("This name is far too long to be valid");**

**});**

**assertEquals("Name must not be null and must be no longer than 20 characters.", exception.getMessage());**

**exception = assertThrows(IllegalArgumentException.class, () -> {**

**task.setName(null);**

**});**

**assertEquals("Name must not be null and must be no longer than 20 characters.", exception.getMessage());**

This helped streamline the testing process, making it more efficient and easier to maintain.

I primarily employed unit testing using JUnit throughout this project. This technique focuses on testing individual methods or functions to ensure they behave correctly in isolation. Additionally, I used boundary testing to ensure that inputs like string lengths and dates fell within valid ranges, especially for fields like task names and appointment dates.

While unit testing and boundary testing were effective, I did not use integration testing or system testing in this project. Integration testing involves testing multiple components together to ensure they work seamlessly. This would be valuable in larger projects where services depend on each other. I also did not use white-box testing, which involves testing the internal structure of the code, focusing on paths, conditions, and loops.

Unit testing is invaluable for ensuring that small units of code function correctly, especially in Agile development environments. Boundary testing is useful in any situation where specific limits must be enforced, such as field length or number constraints. Integration testing is essential in systems where services or modules need to interact, ensuring that they work together as expected.

During the project, I adopted a mindset of caution and attention to detail, especially when testing complex interdependencies between fields. For example, ensuring that the Appointment Service correctly handled date constraints involved understanding how time zones could affect the validation of future versus past dates. I approached testing with the understanding that even small changes could have a significant impact on the program’s behavior.

To avoid bias, I treated each test as if I were a new developer or a user unfamiliar with the code. This helped me identify possible flaws or areas where users might input invalid data. For instance, when testing the Task Service, I imagined how users might input invalid or blank task names and ensured the code could handle those cases. Basically, trying to account for accidental input that could happen or even malicious types of input.

Maintaining discipline in testing is essential for producing high-quality software. Cutting corners in writing or testing code can lead to situations where short-term solutions create long-term problems. Throughout this project, I made sure to thoroughly test each feature before moving on to the next. For example, in the Contact Service, by enforcing strict checks on phone numbers and names, I prevented future issues related to incorrect input formats. As I continue in this field, I plan to avoid problems such as these by consistently working to adhere to best practices and maintaining the most comprehensive test coverage I can.