CS320 Project Two: Summary and Reflections Report

Hunter Lucas

CS320 Software Test, Automation QA

Professor Angel Cross

April 19, 2025

In the process of developing the mobile application for Grand Strand Systems, I utilized unit testing to ensure the functionality of key features. I focused on three core services of the application, the appointment service, contact service, and task service. For each of these services, I wrote JUnit tests to check the basic functionalities. Adding, updating, deleting, and retrieving records. These tests made sure that all the operations worked according to the specified requirements.

For the appointment service, the AppointmentServiceTest.java file was created to test the functionality of scheduling appointments. I wrote tests to confirm that appointments could be successfully added and deleted. For example, in the testAddAppointment() method, I ensured that an appointment with a unique ID was added successfully. In testDeleteAppointment(), I checked that deleting an appointment resulted in the appointment no longer being accessible. I also tested boundary conditions like attempting to add a duplicate appointment ID, which was handled in testAddDuplicate(). This made sure that the need to enforce unique identifiers as part of the requirement. For the task service, in the TaskServiceTest.java file, I focused on adding, updating, and deleting tasks. A key test case, testAddTask(), ensured that a task could be added correctly with valid attributes, while testDeleteTask() confirmed that tasks could be deleted. I also tested handling invalid task IDs in testAddDuplicateTask(), which ensures that duplicate IDs are not allowed. These tests ensured the system adhered to the task management requirements, with checks for both correct task creation and proper exception handling. For the contact service, the ContactServiceTest.java and ContactTest.java files were written to validate the contact related operations. I checked that contacts could be created, updated, and deleted, and I made sure that invalid IDs or improper inputs caused exceptions. For example, testAddContact() ensured that a valid contact was added, while testUpdateFirstName() confirmed that changes to

attributes like the first name were reflected correctly. These tests aligned with the requirement that customer data should be properly stored and manipulated.

My testing approach was aligned with the software requirements in that each test was crafted to ensure the application met its functional expectations. For example, in appointment management, the system required that appointments had unique IDs and could be added or deleted. In AppointmentServiceTest.java, I included tests like testAddDuplicate() to ensure the application handled the unique ID requirement. Similarly, the tests for task management and contact management adhered to the constraints on valid task IDs and customer attributes, like name length and phone number formatting.

The overall quality of the JUnit tests was ensured by the coverage they provided across the critical functionalities. The tests checked not only the basic use cases (adding, updating, and deleting) but also edge cases, such as adding duplicate records or passing invalid input. The test coverage percentage was 100% for each test. I made sure that the application would behave as expected in production. The effectiveness of these tests was evident in the fact that they caught potential errors early in the development process and ensured the integrity of the application's core functionalities.

Writing the JUnit tests was an iterative process that involved a bit of planning, testing and frequent adjustments. I started by identifying the main features that needed testing, then wrote tests that covered both the basic functionalities and edge cases. As I wrote each test, I ensured that the assertions were simple and clear. For example, in testAddAppointment(), I used assertEquals(appointment, service.getAppointment("HL1028")); to directly test whether the appointment with ID "HL1028" was added successfully. These assertions ensured that the updated task reflected the correct changes, confirming that the application was behaving as

expected.

To ensure that my code was technically sound, I regularly ran the JUnit tests and reviewed the results. Any failing test was addressed immediately, and I ensured that all exceptions were handled properly in the code. For instance, in testAddDuplicateTask(), I tested for the IllegalArgumentException by using assertThrows(IllegalArgumentException.class, () -> taskService.addTask("task1", "Another Task", "Should be another description."));. This line verified that the system correctly threw an exception when attempting to add a duplicate task.

During this project, I employed several software testing techniques, including unit testing, boundary value testing, and equivalence class testing. Unit testing was the core technique used for testing the individual features of the application. This method focused on testing isolated units of functionality, like adding, updating, and deleting tasks or appointments. It helped identify and fix issues in specific methods early in the development cycle. Boundary value testing was employed when testing limits, such as the maximum length of descriptions in appointments or tasks. This type of testing helped validate that the application correctly enforced constraints. Equivalence class testing was used to ensure that valid inputs were accepted and invalid ones were rejected. This ensured that only valid inputs were processed.

There are other testing techniques I did not employ for this project, namely integration testing and exploratory testing. Integration testing is used when testing how different parts of a system interact, such as integrating a database with a web application. Since the project did not involve interactions between multiple systems, integration testing was not needed. Exploratory testing involves manually testing the application without predefined test cases to uncover hidden bugs. This was not necessary for this project since the focus was on individual features, and the application did not have a user interface that required unscripted exploration.

When approaching this project as a software tester, I used a mindset of curiosity and attention to detail. I knew that a single mistake in the code could lead to failures in other parts of the system, so I made sure to test each feature thoroughly. As a software developer, I can see how bias and confidence could be a concern when testing my own code. It would be easy to overlook potential issues or assume everything works correctly. That's why I intentionally tested for edge cases, such as invalid IDs and overly long descriptions.

Being disciplined in maintaining quality is essential when developing software. Cutting corners in code or tests can lead to technical debt, which makes future development more difficult and prone to error. Throughout this project, I ensured that my tests were comprehensive and covered both normal and edge cases. In my future practice, I will continue to prioritize quality over speed, writing tests that ensure the system works as expected and reducing the risk of future issues.