Christopher Keenan

04/20/2025

Project Two Summary Report

CS-320

### **Summary**

**Extent of Alignment to Requirements**  
 During development of this project, every test was specifically aligned with the described feature requirements. For example, the task object rejected task IDs longer than 10 characters, names over 20 characters, and descriptions over 50 characters. My unit tests included boundary tests for each of these constraints. One specific example of this is assertThrows for inputs that violate guideline limits.

**Defending Quality of Tests and Coverage**  
 The coverage of my tests included the breadth and depth needed to ascertain that each class was tested for both valid and invalid inputs. For example, in ContactServiceTest, I verified successful contact creation and included exceptions when inserting duplicate IDs.

Using assertions like assertEquals, assertNotNull, and assertThrows allowed me to validate both the expected outputs and handle errors appropriately. Based on a manual review, I would estimate over 90% coverage of logic based on manual inspection of all code paths. One aspect I didn’t catch initially was leaving a setter in which affected the mutability of the ID field. This would be the only reason I didn’t achieve 100% test coverage.

**Unit Testing Approach for Each Feature** In order to address all of the three core features of this mobile service, including Contact, Task, and Appointment, I implemented a structure based unit test approach which utilized the JUnit framework. The main goal was to confirm objects were successfully created, test functionality of all service methods, and validate input.

* *Contact Feature*: I validated object field constraints including values set to null, length restrictions on characters, and checked that correct value retrieval was occurring via getters like getContact. I also tested create, read, update, and delete operations within the ContactService class.
* *Task Feature*: My approach here was similar to the contact object, I validated task object data rules were followed through verifying null values if found with a strong emphasis placed on the uniqueness of the task ID and field updates.
* *Appointment Feature*: I ensured required behavior of the appointment object creation and all relevant service operations such as adding, deleting, and utilizing getters to retrieve information, attempting to focus on immutability of ID and validating appointment date.

**Experience Writing the JUnit Tests** I would say the tests helped me internalize the design principles behind each class. I ensured the technical soundness of my application by following these best practices for software development:

* Used consistent setup methods to reduce redundancy.
* Included tests for both positive and negative outcomes such as successful contact addition versus adding a duplicate ID, respectively.
* I reused data inputs that had unique values in an attempt to simplify error tracing.

One example of this would be in AppointmentTest, as the following test ensures complete object instantiation:

*assertThrows(IllegalArgumentException.class, () -> new Appointment("01234567890", date, "Description"));*

This helped verify that task IDs beyond 10 characters are not accepted by the application.

**Efficiency**:  
 The service tests are modular in nature and I reused task/contact/appointment instances when applicable. This made all of the tests efficient and readable without redundancy. One example from the code would be in appointment\_test and contactservice\_test, where each aspect of the base class functionality was unit tested (testGetAppointmentDate, testUpdateDescription, testUpdatePhone, testUpdateAddress, etc).

### **Reflection**

**Software Testing Techniques Used**

* *Black-box testing*: I wrote all unit tests from the perspective of an external user. I avoided modification of the internal components of each implemented class and prioritized validation of public method behavior.
* *Boundary testing*: I placed a great emphasis on upper-limit input sizes (e.g., 10-char IDs, 20-char names).
* *Negative testing*: I intentionally passed invalid arguments to validate exceptions were adequately handled.

**Software Testing Techniques Not Used**

* *Integration Testing*: I didn’t test how different services interact.
* *Mock Testing*: No mocks or stubs were needed because the logic was simple and internal to each class.

These techniques were not utilized in the scope of this project as they are better suited for larger systems with layered architecture or services that consume external APIs.

**Mindset** I adopted a high level of attention to detail and utilized a cautious mindset while testing. I carefully isolated each test case and ensured all boundary conditions were addressed. For example, in ContactTest, I ran tests with exactly 10-character IDs to ensure upper-bound compliance. It is important to appreciate the complexity and interrelationships of the code being tested in all use cases as these considerations are necessary for preventing economic losses due to issues arising in production from lack of extensive test coverage.

**Limiting Bias** Even though I developed the classes myself, I made sure to challenge each method with both expected and invalid input to simulate possible edge cases of end user behavior. I did not assume that the code would simply "just work." For instance, I specifically tested for null values even though I had written null checks within the constructor. Despite this focus on ensuring full test coverage, I can definitely envision bias being a relevant factor in the possibility of a developer testing their own code. No one wants to admit the code they spent effort and time compiling could possibly have issues. Therefore, it is important to have code inspected by someone who did not directly contribute to the implementation.

**Discipline/Commitment to Quality** Writing tests first (or closely alongside development) prevented me from taking shortcuts. I avoided omitting validation just to get the code base operational. One specific example was in TaskService where I implemented ID uniqueness validation early on in the development process, as it prevents test failures later in the lifecycle. In the future, I plan to adopt test driven development more often, use code coverage tools to quantify and assess any gaps, and incorporate static code analysis to catch possible regressions.