Zack A. Lodi

CS - 320

**Summary and Reflections Report**

**Summary**

a) Describe your unit testing approach for each of the three features.

* *To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.*

- When testing the **Contact** feature, I focused on making sure each part of the contact followed the rules in the requirements. I checked that contact IDs were no longer than ten characters, names and addresses weren’t too long or null, and phone numbers had exactly ten digits. I also tested edge cases like using the maximum allowed lengths to make sure the system didn’t reject valid inputs.

assertThrows(IllegalArgumentException.class, () -> {

new Contact("12345678901", "John", "Doe", "1234567890", "123 Main Street");

});

- For the **Task** feature, I made sure that the task ID, name, and description followed the same kind of rules. I tested that null values and strings that were too long were rejected, and that valid updates worked properly. I also included edge cases to make sure the system accepted valid inputs at the limit and rejected anything beyond.

assertThrows(IllegalArgumentException.class, () -> {

new Task("123", null, "Finish homework");

});

- The **Appointment** tests were built around the rule that appointments must have a valid ID, a future date, and a short description. I tested that past dates and null values were rejected, and that future dates were accepted. I also checked that long but valid descriptions worked.

Calendar cal = Calendar.getInstance();

cal.add(Calendar.DATE, 1);

Appointment appt = new Appointment("A123", cal.getTime(), "Doctor visit");

assertTrue(appt.getAppointmentDate().after(new Date()));

* *Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?*

- The **Contact** tests were strong because they covered a wide range of situations. I didn’t just test the normal cases, but also included invalid inputs and edge cases. I used *assertThrows* to make sure errors were triggered when needed, and *assertEquals* to confirm that valid data was stored correctly. Therefore I believe the Contact feature was reliable and that the tests provided solid coverage of the expected behaviors.

Contact contact = new Contact("1234567890", "John", "Doe", "0123456789", "123456789012345678901234567890");

assertEquals("John", contact.getFirstName());

- The **Task** tests were effective because they were simple, focused, and covered all the important behaviors. I tested creating tasks, updating them, and handling bad inputs. I also made sure that the service class rejected duplicate IDs and threw errors when trying to update tasks that didn’t exist. By covering both successful and failing scenarios, the Task feature worked as intended and handled errors properly.

Task task = new Task("123", "Homework", "Finish problems");

task.setName("Chores");

task.setDescription("Clean the kitchen");

assertEquals("Chores", task.getName());

- The **Appointment** tests covered all the key rules and included edge cases. I tested both good and bad inputs, and made sure the service class handled duplicates and deletions correctly. Using *assertThrows* and *assertTrue* helped confirm that the validation logic was working and that the system wouldn’t accept invalid appointments. This approach helped ensure the Appointment feature was dependable and met the project’s requirements.

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("A1", null, "Meeting");

});

b) Describe your experience writing the JUnit tests.

* *How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.*

- To make sure my tests were technically sound, I wrote them to match the rules in the code exactly. I used assertions to check that errors were thrown for bad inputs and that updates worked correctly. I also made sure to test each method with both valid and invalid data to confirm that the logic worked in all situations.

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("A1", futureDate, null);

});

* *How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.*

- I kept my tests short and focused so that each one only tested one thing. This made it easier to find problems when something went wrong. For example, I had separate tests for updating fields with bad data, which helped me quickly spot issues. I kept the tests organized and specific, making them easier to read, maintain, and debug.

assertThrows(IllegalArgumentException.class, () -> task.setName("ThisNameIsWayTooLongToBeValid"));

**Reflection**

a) Testing Techniques

* *What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.*

- In this project, I mainly used three types of software testing techniques: **boundary testing, negative testing, and functional testing**. Boundary testing helped me check the limits of what the system should accept. I tested inputs that were exactly at the maximum allowed length, like a contact address with exactly 30 characters or a task description with exactly 50 characters. This made sure the system didn’t reject valid inputs just because they were at the edge. Negative testing was about trying to break the system by giving it bad data like null values, overly long strings, or invalid phone numbers. These tests helped confirm that the system would throw errors when it should. Functional testing was used to make sure the main features worked correctly. I tested adding, updating, and deleting contacts, tasks, and appointments to confirm that the services behaved the way they were supposed to.

* *What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.*

- Some testing techniques I didn’t use in this project include **integration testing, regression testing, and performance testing**. *Integration testing* checks how different parts of a system work together, like connecting a database to a user interface. I didn’t need this because my project focused on isolated service classes that didn’t interact with external systems. Regression testing is used to make sure new changes don’t break existing features. Since this was a one-time build and not an ongoing project with updates, regression testing wasn’t necessary. Performance testing checks how well the system handles heavy use, like lots of users or large amounts of data. My project didn’t involve high traffic or big datasets, so performance wasn’t a concern.

* *For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.*

- Each of these techniques has its place depending on the type of project. Boundary testing is useful in systems with strict input rules, like banking apps or medical software, where any small mistake can cause big problems. Negative testing is important for security and reliability; it helps ensure the system won’t crash or behave badly when users make mistakes. Functional testing is needed in almost every project to confirm that features work as expected. Integration testing becomes important when different parts of a system need to work together smoothly, like in web apps that connect to APIs or databases. Regression testing is important in long-term projects with frequent updates, so developers can be confident that new code doesn’t break old features. Finally performance testing is critical for apps that need to be fast and scalable, like social media platforms or e-commerce sites.

b) Mindset

* *Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.*

- While working on this project, I took a fairly cautious approach to testing. I didn’t assume that the code would work, and I tried to think like a user who might make mistakes or enter bad data. For example, I tested what would happen if someone tried to schedule an appointment in the past or entered a phone number with fewer than ten digits. These are things that could actually happen in real life (not just edge cases). I also paid close attention to how different parts of the code depended on each other. In the service classes, deleting or updating a contact required a valid ID. If that ID didn’t exist, the whole operation could fail. So I tested for those situations too.

assertThrows(IllegalArgumentException.class, () -> {

service.deleteContact("999");

});

* *Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.*

- Since I was testing my own code, there was a risk of bias. To avoid that, I made a point of writing tests that tried to break the system. I tested null values, overly long strings, and invalid formats. I passed a null name to a task update method just to see if the system would catch it. I also used *assertThrows* a lot to make sure errors were actually being thrown when they should be. This helped me stay honest and not overlook problems.

* *Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.*

- Being disciplined about testing is one of the most important parts of writing good software. It’s tempting to skip tests or cut corners when you’re in a rush, but that can lead to problems that pile up and get harder to fix over time. I made sure to write full tests for each feature, even if some of them felt repetitive (I tested both valid and invalid inputs for every field in the Contact, Task, and Appointment classes). I also tested service-level operations like adding and deleting, even though the logic seemed simple. This kind of thoroughness helps catch bugs early and keeps the codebase clean.

- In the future, I plan to avoid potential issues by sticking to good habits like fail-fast validation, clear error handling, and strong test coverage. Taking the time to test properly now saves time and stress later.

assertThrows(IllegalArgumentException.class, () -> service.addTask(task2));