Reece Ebel

CS 320

**Project Two**

**Summary**

For each of the three services (contact, task, and appointment) I implemented a very thorough unit testing approach using JUnit. The focus was to isolate each component of the service and testing each independently to ensure that it met the specific requirements. For example, in the contact service, I created tests which validated the creation of contact objects, ensuring that constraints on fields like contact ID, first name, and phone number were strictly enforced. Similarly, in the task and appointment services, I wrote tests to verify that tasks and appointments could be correctly added, updated, and deleted, adhering to the defined constraints.

**Alignment to Software Requirements**

My testing approach was very closely aligned with the software requirements provided by the customer. Each test case was designed to verify that the service behaved as expected under various conditions, edge cases included. As an example, in appointment service, the requirement was that the appointment date could not be in the past, and the appointment ID could not exceed 10 characters. I created tests which checked for these conditions, ensuring that any input violating these constraints would result in an error. This is evident in the ‘AppointmentTest’ class, where I used the below test to validate the appointment date.

A computer code with text

Description automatically generated

**Effectiveness of Junit Tests**

The overall quality of my Junit tests was high, as reflected in my comprehensive coverage. The tests were designed to cover all possible scenarios, including valid/invalid inputs, edge cases, and error handling. By ensuring that each method was tested under various conditions, I could confidently assert that the services would behave as expected in production. The coverage percentage was a strong indicator to the efficacy of these tests, as it showed that most, it not all, of the code was exercised during testing. This thorough approach helped minimize the risk of bugs that may appear later on.

**Experience Writing JUnit Tests**

Writing JUnit tests was a great learning experience for me which helped to reinforce the importance of technical soundness and efficacy in programming. I ensured that my code was technically sound via a rigorous testing of all branches of logic, using assertions to validated expected outcomes. An example of this is in the ‘ContactServiceTest’ class, I wrote the below test to ensure that the ‘addContact’ method correctly handled the addition of a contact with a unique ID.

A screenshot of a computer

Description automatically generated

To assure the efficiency, I used in-memory data structures such as ‘HashMap’ to store contacts, tasks, and appointments, enabling fast lookups and minimizing the computational overhead. For example, the ‘ContactService’ class utilized a ‘HashMap’ to manage contacts efficiently.

A close up of a text

Description automatically generated

This data structure ensures that operations like adding, updating, and deleting contacts were performed in constant time (O(1)), which enhances overall performance.

**Reflection**

**Testing Techniques Employed**

In this project, I mostly employed unit testing as the software testing technique. Unit testing involves testing the individual components of software, in isolation, to verify that they function correctly. The characteristic of unit testing is its focus on small, manageable pieces of code, ensuring that each piece of the whole behaves as expected prior to integration with other components. This approach is was highly effective for the contact, task, and appointment services, as it allowed me to identify and fix issues early in development.

**Other Testing Techniques**

While unit testing is the primary technique I used, there are other software testing techniques that I did not use in the project, like integration testing, system testing, and acceptance testing. Integration testing is focal on verifying interactions between components, or systems, to ensure they work correctly together. This testing method is important in more complex applications where multiple services or modules interact. System testing involves testing the entire system, often against the requirement specifications, which assures that it meets expected behavior. Acceptance testing, alternatively, involves testing the software from a user’s perspective to make sure that it meets their needs and expectations.

**Users and Implications of Techniques**

Each testing technique has its practical uses and implications. Unit testing is most appropriate early on in development when individual components are still being developed and refined. It helps to catch bugs early on and reduces cost and time required to fix issues later in the development process. Integration testing is more appropriate once multiple components have been developed and need to work together. It helps in identifying issues related to data flow, communication, and dependencies between components. System testing is best for the final stages, where the entire system is tested to ensure it meets the requirements and performed will under real-world conditions. Acceptance testing is best for ensuring that the final product meets the user’s needs and expectations.

**Mindset**

**Caution in Testing**

Throughout this project, I remained cautious and conscious to recognize the complexity and interrelations of the code I was testing. It was important to acknowledge that small changes in one part of the code may cause unintended consequences somewhere else. An example of this is while testing the task service, I was careful in my considerations on how updates to a task details may affect other parts of the application. My cautious mindset helped me to identify potential issues early and ensure that the service worked correctly in all scenarios.

**Limiting Bias**

To limit bias in testing, I remained conscientious of the user, even another developer’s, perspective. I recognized that as the developer of the code, I may overlook issues that someone else may notice. To counter this problem, my code was reviewed critically by my super awesome professor and I relied on his feedback when possible. This approach aided in maintaining objectivity and ensure the tests were comprehensive.

**Discipline and Commitment to Quality**

Being disciplined in my commitment to quality was very important for this project. Cutting corners when it comes to writing, or testing code, could lead to technical debt and future maintenance issues. To avoid this, I made sure to write thorough tests for each feature and refactor the code where necessary to improve the readability and maintainability. I avoided hard-coding values in the tests and instead used constants which made the tests more robust and easier to update if the requirements changed. This helped make sure that the code was of good quality and easier to maintain.