CS 320   
Max Day

**Unit Testing Approach for Each Feature**

**Contact Service:** My unit testing approach for the Contact Service was implemented by creating individual test cases for operations, like adding, updating, and deleting contact records. Within the ContactServiceTest class, I used a HashMap to simulate a repository of each contact ‘object’. The tests helped to ensure that each contact had unique IDs, and validated attributes like the first name, last name, phone number, and address. All of the setup methods annotated with @Before initialized necessary data structures, providing consistency across all test cases. Task Service unit testing approach I had focused on validating task creation and task management features. The TaskServiceTest class leveraged JUnit’s JUnitCore to run multiple test scenarios. Functionality, such as task name validation and deadline management, was covered to ensure tasks adhered to the requirements. My Appointment Service testing was focused on verifying appointment creation and scheduling functionalities.

The testing approaches I implemented aligned well with the requirements for Contact and Task Services – this is really seen by the robust validations and consistent test setup. An example; the ContactServiceTest includes tests to verify constraints such as things like maximum character limits for names and valid phone number formats..

The quality of JUnit tests can be evaluated based on their co ,lpverage and reliability. The tests for Contact and Task Services achieved high coverage, ensuring that edge cases were tested. For example, the ContactServiceTest effectively tested scenarios where duplicate contact IDs were generated, ensuring robustness..

**Experience Writing JUnit Tests**

Writing JUnit tests was a learning-intensive experience that emphasized technical precision and efficiency. Specific assertions, such as Assert.assertEquals() and Assert.assertNotNull(), were used to validate critical operations. For example, in ContactServiceTest, the line:  
Assert.assertEquals("John", contactservice.getContact(uniqueID).getFirstName());  
validated that the contact retrieval function returned the correct name for a given unique ID. Efficient testing was achieved by reducing redundancy in setup. The @Before method ensured that data initialization was reused across test cases, saving time and improving consistency. For example:  
@Before

public void setup() {

contactservice = new ContactService();

contact = new Contact("123", "John", "Doe", "1234567890", "123 Elm St");  
contactservice.addContact(contact);

}  
This setup streamlined multiple test scenarios.  
**Employed Techniques:**  
I used focusing on Unit testing and Black Box testing. Unit testing focusing on validating individual methods for each service. It also incorporated isolated testing, for using mock data and assertions to validate expected outcomes. Black Box Testing helped to ensure that the services adhered to requirements without examining internal code structure. This helped with input-output validation and test case design based on functional requirements.

**Techniques I did not use:**

Integration testing & Regression Testing. Integration testing technique would test interactions between services (e.g., Contact and Appointment). This would help to focus on data flow and dependency validation. Regression testing helps to ensures that changes in code do not introduce new bugs. This requires a suite of test cases for repeated execution. I feel Unit testing is the most ideal for validating isolated functions in backend services. Integration testing makes sure that modules interact with each other correctly and regression testing is more important in projects with constant updates for stability in the long term.   
I tried to use caution when validating edge cases. An example: In ContactServiceTest, Imade sure adding a contact with a duplicate ID raised an exception. This appreciated the complexity of maintaining data integrity in interconnected services.

To make sure I adhered to minimizing bias, I used ContactServiceTest, I ensured that adding a contact with a duplicate ID raised an exception. This approach appreciated the complexity of maintaining data integrity in interconnected services.

**Limiting Bias:** To minimize bias, I adhered to a structured test-writing process, treating the code as a black box. Reviewing my own code with a critical mindset was essential. For instance, I deliberately designed tests to fail under invalid input conditions, ensuring comprehensive coverage.

**Discipline and Quality Commitment:** A disciplined approach to testing is vital to avoid technical debt. Cutting corners compromises software reliability and increases maintenance costs. For example, by writing modular and reusable test setups, I avoided repetitive code, improving long-term maintainability. My plan to avoid technical debt includes adopting automated testing tools and maintaining detailed documentation for all test cases.

a structured test-writing process, treating the code as a black box. Reviewing my own code with a critical mindset was essential. Example; I deliberately designed tests to fail under invalid input conditions. For a disciplined approach to testing is very important to avoid technical uncertainty and costs. Cutting corners compromises software reliability and increases maintenance costs. For example, writing modular and the same tests; I avoided repetitive code, improving long-term maintainability. My main goal is to avoid any technical debt and adopting the automated testing tools I have learned as well as maintaining detailed documentation for all of my future test cases.

In conclusion, doing this project made me realize how important it is to align testing strategies with software requirements and maintaining quality assurance. Going forward, I aim to enhance my testing strategies by incorporating integration and regreaaiong more in my testing to understand more methods and have a comprehensive software toolset.