Unit Testing Approach:

When it came to testing the mobile application project, I took a thorough and careful approach to make sure the reliability of each part of the project. I used unit tests to verify the accuracy of individual components, specifically the Task, TaskService, and other classes. The reason why I’ll be using examples from Task and Task Service is because I’m essentially just applying the same logic used in those first two classes and tests in all the other classes. It’s also the first part of my project, and feel like that first assignment really laid the foundation for everything else.

Task Class:

Constructor Validations: I made sure to test the Task class constructor so that it properly validated inputs such as task ID, full name, and description. For example, the test validConstructor checked for valid inputs, while tests like nullIDConstructor and invalidIDConstructor checked how the constructor handled invalid inputs.

Setter Methods: I also created tests to validate the setter methods. For example, the test nullNameConstructor checked if the setter for the name field threw an exception when given a null value, ensuring that the method was robust.

TaskService Class:

Add Task: The testAddUniqueTask verified that the functionality of adding a new task worked correctly, making sure that the task was stored properly in the tasks list.

Delete Task: The testDeleteTask confirmed that tasks were correctly removed from the tasks list.

Update Task: To make sure that only existing tasks were updated, I validated the update functionality with for example the testUpdateTask for valid IDs and testBadUpdateTask for invalid IDs.

Alignment with Software Requirements:

My testing approach closely aligned with the software requirements, which emphasized the need for robust validation of appointment/task/contact and appointmentservice/taskservice/contactservice by thoroughly validating IDs, full names, appointment times, contact info, and descriptions in my project. I also made sure the integrity and reliability of the application was tested. The checks in the Task class for example (validateID, validateName, validateDescription) directly address the requirement of ensuring valid task properties.

Quality of JUnit Tests:

The quality of my JUnit tests can be justified by looking at their coverage and how well they handle edge cases. I made sure to cover all the important parts of the project, such as:

Positive Cases: I tested valid inputs to make sure the normal functionality was working properly.

Negative Cases: I tested invalid inputs and edge cases to make sure that the application handled errors gracefully.

For instance, the testAddUniqueTask not only checked if a task was added, but also made sure that the size of the task collection was updated correctly. Similarly, testDeleteTask and testUpdateTask included assertions that confirmed the absence and presence of tasks after performing operations.

Experience with Writing JUnit Tests

Writing JUnit tests was a valuable learning experience that highlighted the importance of thinking about potential points of failure. I made sure to follow important rules to make sure my program ran smoothly, such as:

Using Assertions: For example, assertEquals(fullName, tempTask.getFullName()) made sure that the values matched the expected results.

Testing Edge Cases: By including tests like nullIDConstructor and invalidIDConstructor, I made sure to cover edge cases.

Code Efficiency: By keeping the tests concise and focused, like assertTrue(TaskService.tasks.containsKey(id)), I made sure efficient execution.

Testing Approaches

Black Box Testing:

Black Box Testing focuses on input-output relationships without delving into the internal code structure. Used to verify outputs based on given inputs. It’s perfect for validating user-facing functionalities where the implementation details are hidden.

White Box Testing:

White Box Testing examines the internal mechanisms or operations of an application, including specific code paths, conditions, and branches. It’s crucial for making sure that the internal processes of a function or method perform as intended.

Boundary Testing:

Boundry testing examines the boundary values of input ranges to confirm that the application handles extreme cases correctly. It’s used for areas like taskID and fullName where input length restrictions are defined.

Other Approaches:

Integration Testing:

Integration Testing examines the communication between integrated units or modules. It’s valuable for confirming the communication between Task and TaskService.

System Testing:

System Testing evaluates the entire integrated software to validate compliance with specified requirements. It helps confirm that the entire application functions as expected in a real-world setting.

Regression Testing:

Regression testing makes sure that new code modifications do not negatively impact existing features. It’s vital for continuous development to prevent updates or fixes from disrupting previously functional features.

Mindset

Maintaining a careful and thorough mindset was really important for the project. As someone that’s learning software testing, understanding the details and connections within the code helped find potential issues and unique scenarios. For example, figuring out the link between Task validation methods and TaskService operations helped me to create better tests.