**Software Testing – Summary and Reflection**

Alignment to Requirements:

As a software engineer of Grand Systems, I need to develop and create unit test cases for mobile application, which consists of Contact, Task, and Appointment services. The requirement for the Contact class is to have id, firstname, lastname, phone, and address fields, and ContactService class should be able to add, update, and delete contacts. Next, the requirements for the Task class are to have an id, name, and description, and the TaskService class should provide the ability to add, update, and delete tasks. Finally, the Appointment class consists of id, date, and description fields, and the Appointment service offers features to add and delete appointments.

Each field and operation has validation to implement, like not null, max length, and date should not be in the past. All the base classes should check the uniqueness before performing add, update, and delete operations. So, the id field should be used to check the uniqueness of a particular object and not be allowed to update the id field with a different value. To enforce this, POJO classes do not have a set method for the id field, so there is no way to update the id after the object is created.

Effective Tests:

The unit testing goal is to test each line of code with various possible scenarios and data. Able to achieve 99.8% of overall code coverage. Each main class has its own Junit test class, and each class has test methods for each scenario. E.g., one method to test id field not null validation and another method to test id field max length validation. So, in this approach, we can narrow down the issue to a particular field and validation rule instead of combining scenarios in one test method.

Also, test cases should be formed to test both positive and negative cases and the edge cases. Negative and edge cases commonly have defects because positive scenarios are primarily focused on and tested in earlier stages. For instance, the integer id field can also accept negative values, which should be validated through a test case if it is not allowed.

Technically Sound Code:

Test cases are well-defined without any duplication and easy to understand and maintain. Also, the reusable code moved to a helper method, which can be called in individual test cases, so if we need to make a change, it should be in one place, which helps to modify and test changes quickly. E.g., the getContact() method from the Contact test class is called in multiple test case methods to initialize the contact object with default values.

Before and after test case execution steps are implemented to increase the test case performance and for memory resource management. @BeforeAll annotation used to setup configuration before each test case execution and it is defined once instead of defining in each test case method. @AfterAll annotation helps to destroy any resources used for test case execution at the end.

Efficient Code:

An annotation-based framework from Jakarta was used in test cases to test field validation rules. It can save several lines of code and make it easy to understand the validation rule by looking into the annotation name and validation error message, as mentioned in the code below.

*@NotNull(message = "appointment id should not be null")*

*@Length(max = 10, message = "appointment id length should not exceed 10")*

***private*** *String appointmentId;*

These annotations are from the standard library, which already has validation proof to prevent defect escape from custom code. Each field validation has one valid and one invalid test case, so if it fails, we will know whether the positive or negative scenario failed; method names are suffixed with \_valid or \_invalid words to identify the scenario quickly.

Techniques Employed:

The JUnit framework (annotation-based) helps test each line of code using assertion techniques like comparing actual output (derived from existing code execution) with the expected output (known fixed value). For example, the assertEquals method has two arguments; the first argument is the expected value, and the second one is the actual value. This method will return true if both values are the same; otherwise, it becomes false. The test is successful if the assert method returns true and code logic is working as expected; in case of false, logic has an issue that must be verified and fixed.

Jakarta validator libraries are used in test classes; in POJO classes, attributes are defined with validation annotations with error messages to throw in case validation is unsuccessful.

Other Techniques - Usage and Implication:

Numerous testing frameworks are used in Java, like Mockito, Selenium, Cucumber, PowerMock, etc., which are not used in this project. Mockito will be helpful if we want to mock any object with default values to test the code/functionality. One of the best examples to use is this in API controller testing. Selenium is another popular framework for web application automation testing. Cucumber is a behavior-driven development (BDD) testing framework using natural English to write test cases. PowerMock is the same as Mockito used to mock objects, but it can provide deeper mocking capabilities like byte code manipulation and class loader options.

I have not used the assertThrows feature from JUnit5 in case of validation failures. The reason for this is I am using a validator framework, which will give all the constraint errors in a collection set that can be iterated and decide if a particular validation fails.

Caution:

We must be more cautious about similar requirements, e.g., each service has essential add, update, and delete operations. But in the Appointment service, no update operation is specified in the requirements, so it doesn’t need to be implemented and tested. Another example is the different max length limitation for each field, so re-verify each field's max length value and ensure the same got implemented in the main class and test cases. Caution is required to avoid errors in any copy-paste code for similar functionality/test cases. Another common issue is typo errors, which can be found and fixed by self-review and/or peer reviews.

Bias:

Bias usually occurs when you review test cases for the code you wrote. Requirements should be reviewed in the implementation code and the test cases without bias or ambiguity. Sometimes, to test single implementation methods, we must write more test cases based on the number of positive and negative scenarios. Test data plays an essential role in testing different scenarios, and make sure no bias exists when reviewing the test data, combination, and scenarios. For example, the populateTaskMap() method sets initial tasks in the in-memory data collection map before testing individual test cases in TaskService. There are three tasks populated with task id starting from 1 to 3, so to add a new task in the positive test case, we can use 4 as the task id. So, the initial data setup impacts the actual positive and negative test case data, which should be reviewed thoroughly.

Discipline:

When writing test cases, we need to get maximum coverage to avoid any defects shipped to production, which can cause financial and reputation losses. The requirement asks to cover a minimum of 80% code coverage, but we must push it forward to meet the maximum instead of the minimum. I can manage to get 99.8% overall coverage. It is better to get maximum coverage for individual test classes rather than overall coverage; e.g., initially, some of the get methods are not covered when we test the Contact service class, and I have made sure to check all the attributes using its corresponding get methods in the updateContact test method.