7-2 Project Two – Summary and Reflections Report

Summary:

I developed Contact, Task and Appointment service for the mobile application for a customer. During this development I employed unit testing approach to ensure that all the codes are tested, all functionality is covered, all boundary cases are covered, and all the requirements are met. I used JUnit tests for unit testing.

Alignment to the Software Requirements: My approach to development and testing was closely aligned to the requirements provided.

Contact Service: For Contact service I focused on creating contact, adding contact, deleting contact and updating contact fields (except Contact ID). The unit tests ensured that “null” is not passed and the field values does not exceed the required length. I also incorporated the feedback received to use assertAll().

@Test

**void** testUpdateValidFields() {

Contact contact = **new** Contact("Contact123", "Niyati", "Shetty", "8006681249", "2500 NRiver Rd, Manchester,NH");

contact.setFirstName("SNHU");

contact.setLastName("University");

contact.setPhoneNumber("8006007000");

contact.setAddress("3500 Main Rd, Nashua, NH");

*assertAll*("Test Update Valid Fields",

// test setFirstName()

() -> *assertEquals*("SNHU", contact.getFirstName()),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setFirstName(**null**)),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setFirstName("LargerThan 10 Chars")),

// test setLastName()

() -> *assertEquals*("University", contact.getLastName()),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setLastName(**null**)),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setLastName("LargerThan 10 Chars")),

// test setPhoneNumber()

() -> *assertEquals*("8006007000", contact.getPhoneNumber()),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setPhoneNumber(**null**)),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setPhoneNumber("LargerThan 10 Chars")),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setPhoneNumber("80070060005000")),

// test setAddress()

() -> *assertEquals*("3500 Main Rd, Nashua, NH", contact.getAddress()),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setAddress(**null**)),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> contact.setAddress(

"500 Main Rd, Nashua, NH and more LargerThan 10 Chars"))

);

Task service: I verified that creating, adding the task, deleting the task to in-memory map and updating the description of the task. The test also ensured that ‘null’ is not passed and the fields does not exceed the required length.

Appointment service: I verified creation, addition and deletion of appointments.

For all three services no duplicate service can be added to in-memory map.

JUnit Tests and Coverage: My JUnit test covered all the requirements; the test coverage was 88%. The TaskService coverages was 74%, after I received the feedback, I improved that TaskService coverage to 96%. The test covered Constructor, all methods. I checked for ‘null’ and ensured that the length of the field is enforced.

Experience writing JUnit tests: Using the JUnit tests to test my code was very valuable, with discipline, I was able to test all the requirements and border cases. This approach ensured that my code was technically sound and efficient. In below example, I am ensuring the contact Id is not ‘null’ and it does not exceed l0 character length.

@Test

**void** testContactId() {

*assertAll*("Test Contact ID",

() -> *assertThrows*(IllegalArgumentException.**class**, () -> **new** Contact(

**null**, "Niyati", "Shetty", "8006681249", "2500 NRiver Rd, Manchester,NH")),

() -> *assertThrows*(IllegalArgumentException.**class**, () -> **new** Contact(

"Contact890123", "Niyati", "Shetty", "8006681249", "2500 NRiver Rd, Manchester,NH"))

);

}

Code efficiency: The code written was efficient, in the constructor if the invalid value was passed the exception was thrown. In the below example, each field is validated and for date field, we check the date is not in the past, and then after validation the value is stored.

// Constructor - with 3 attributes

**public** Appointment(String appointmentId, Date date, String description) {

// Validate each field according to requirements

**if** (appointmentId == **null** || appointmentId.length() > 10) {

**throw** **new** IllegalArgumentException(

"Invalid Appointment ID: it cannot be NULL and it cannot be longer than 10 characters.");

}

**if** (date == **null** || date.before(**new** Date())) {

**throw** **new** IllegalArgumentException(

"Invalid Appointment Date: it cannot be NULL and appointment date cannot in the past.");

}

**if** (description == **null** || description.length() > 50) {

**throw** **new** IllegalArgumentException(

"Invalid Appointment Description: it cannot be NULL and it cannot be longer than 50 characters.");

}

**this**.appointmentId = appointmentId;

**this**.appointmentDate = date;

**this**.description = description;

}

Reflection:

Testing Techniques: In this project I employed, black box testing, exception testing, boundary testing and Unit testing. Black box testing helped to focus on the input and output, without knowing the internal workings. Exception testing helped me verify all invalid values are rejected. The Boundary testing helped me to ensure the requirements are met and the input value values are within the range specified by the requirements.

Other testing techniques I did not use in this project are Integration Testing, System Testing, Performance Testing, End to End Testing, Load Testing. We can use the Integration testing to ensure different components work with each other. System testing ensure that entire system works fine. Performance testing is to ensure that system performance during high traffic and it does not crash. End to end testing to ensure that system works fine as expected from start to finish and Load testing is to ensure that system can handle unexpected load, and it does not crash.

Each testing techniques has its own practical uses; the black box testing can be useful to ensure that the system works as expected without knowing the internal implementation. The Code review, static testing can ensure that there are not security holes in the code. The performance and load testing help in ensuring the system performance optimally during high traffic and able to handle unexpected load.

Mindset:

When working on this project I had a cautious mind set, I gave attention the requirements provided and ensured that all the requirements are met. I validated all the fields before creating the object or in the setter functions. The Service objects ensured that no duplicate objects with the same ID is stored in the in-memory map.

During the code review, I did not have any bias, I reviewed the code to match the requirements are met, there are no security holes and all the boundary cases are met. If I was developing and testing my own code, I might have missed some edge cases or not covered all the test cases in my unit tests.

The discipline is very important for a software developer and tester to ensure the quality of the code. With good unit test coverage, we will have minimal bugs going forward, it ensures that all the requirements are met. With good system test we can ensure that all the systems are working as intended. Finding the bug during static testing or in unit testing or system testing will cost less to fix it. Finding bug in later stages of the software development will cost more and it may damage our reputation with customer also, which can lead to fine or compliance violation. My plan to avoid technical debt is by writing the unit tests and system tests with 85% or more coverage, do peer code reviews etc.