7-2: Project Two

Nur Faizah Mas Mohd Khalik

This project required me to utilise a test-driven approach to my development of an application for an unspecified client. The components of this app were a Contact class, an Appointment class, and a Task class, all of which required me to refine and re-write some tests to meet the coverage goals of at least 80% overall. This summary will proceed to break down my testing approach for each class, its efficiency and effectiveness, as well as my overall experience.

For my project, I had to develop an application for a client using a test-driven development methodology. Three classes made up the main parts of the application: Contact, Appointment, and Task. I had to create a set of unit tests for each class in order to achieve the coverage target of at least 80%. This summary will concentrate on my testing of the Contact class, going into detail about the effectiveness and efficiency of each test case I created.

My main goal was to adhere to the demands of each field. In the `ContactTest.java` file, I created the `testContactConstructor` method. To make sure that an `IllegalArgumentException` would be produced if any field was null, I used the `assertThrows` function. This gave me the opportunity to ensure that the object would uphold its integrity from the moment it was created, shielding the class from invalid states.

I started by establishing the guidelines for object creation before moving on to the test case `testValidContact`. In this instance, my goal was to verify that a contact object would be created if the inputs were correct. I was able to verify that the object wasn't null using the `assertNotNull` function, which served as a milestone check for the remaining testing.

I then went on to the `testGetters` function to evaluate the getter methods for the Contact class. Here, I used `assertEquals` to make sure that each getter method had produced the desired results. This was essential in confirming that the retrieval techniques were operating as expected and reaffirming my confidence in the operation of the class.

The setters, which I extensively verified using the `testSetters` method, were the equivalent to the getters. I verified that the appropriate getter function would return the changed data after I used a setter method to update a field by using `assertEquals` once more. This was a crucial check in my battery of tests to make sure that data could be efficiently updated in addition to being retrieved.

```
@Test
public void testContactConstructor() {
    assertThrows(IllegalArgumentException.class, () -> new Contact(null, "First", "Last", "1234567890", "Address"));
    assertThrows(IllegalArgumentException.class, () -> new Contact("ID", null, "Last", "1234567890", "Address"));
    assertThrows(IllegalArgumentException.class, () -> new Contact("ID", "First", null, "1234567890", "Address"));
    assertThrows(IllegalArgumentException.class, () -> new Contact("ID", "First", "Last", null, "Address"));
    assertThrows(IllegalArgumentException.class, () -> new Contact("ID", "First", "Last", "1234567890", null));
}

@Test
public void testValidContact() {
    Contact validContact = new Contact("ID", "First", "Last", "1234567890", "Address");
    assertNotNull(validContact);
}
```

I used the `testSettersException` method to include negative tests because no software component is error-proof. I verified that the setter methods would throw an `IllegalArgumentException` if they were given null values using `assertThrows`. This important safety measure stopped the programme from crashing or entering an unpredictable state when it encountered inaccurate data inputs.

```
@Test
public void testSettersException() {
    Contact contact = new Contact("ID", "First", "Last", "1234567890", "Address");
    assertThrows(IllegalArgumentException.class, () -> contact.setFirstName(null));
    assertThrows(IllegalArgumentException.class, () -> contact.setLastName(null));
    assertThrows(IllegalArgumentException.class, () -> contact.setPhone(null));
    assertThrows(IllegalArgumentException.class, () -> contact.setAddress(null));
}
```

My `testUpdateAndGetCombination`, which combined both getter and setter methods, was more thorough. It was intended to demonstrate how well the setter and getter

methods might complement one another, with one method reflecting the changes made by the other. This test ensured the dependability of the Contact class by giving a more comprehensive picture of how the various parts interacted.

In order to see if I could further push the envelope, I added test cases to cater to edge cases and included exploratory tests that were unscripted. This helped me to see if the app's functionality extended beyond the standard tests that I wrote. The results of these opened my eyes to how I could further improve my development and processes so that I could cover any potential problems that would surface even after the app is released.

Ensuring my tests were not just effective but also efficient was a challenge in itself. I adopted Continuous Integration/Continuous Deployment (CI/CD) principles. I did this by eliminating bottlenecks through identifying problems early in the development process, potentially expediting the overall timeline.

Additionally, I developed specialised tests like `testMultipleUpdates` and `testContactIDSetter` to see if the fields could be updated repeatedly and if the `ContactID` unique identifier could be modified. These were the finishing touches that made sure my lesson was adaptable and flexible in a variety of situations.

```
@Test
public void testContactIDSetter() {
    Contact contact = new Contact("ID", "First", "Last", "1234567890", "Address");
    contact.setContactID("NewID");
    assertEquals("NewID", contact.getContactID());
}
```

Test-driven development has had a big impact on how I approach coding. In addition to ensuring the Contact class's functionality, I've laid a strong basis for future development and debugging by creating these tests for it. This process has given me the opportunity to code with greater assurance, supported by quantifiable metrics that evaluate the reliability

and resilience of my application. I'm also more confident that I can take on bigger projects without worrying about whether or not my code will fall apart.