Project Two: Reflection and Summary

Jorden Thomas

Southern New Hampshire University

Summary

For project one, I used a unit testing approach for each of the three features. I focused on making sure that this approach would meet the software requirements. Making sure to meet requirements such as the ID being updateable, is unique and can be used to locate and change information such as the address can be updated by using the ID to locate the account. This was done by doing a test for all the required functionalities. By also using a variety of testing techniques I was able to ensure that my tests were effective. The JUnit tests were written in a modular fashion which made them easier to maintain and debug, I also used a variety of assertions to verify the results of my test. This would help in making sure that the code was technically sound and efficient.

My focus when creating the Junit tests was to make sure that variables were made based around the requirements. By focusing on my variables being related to the requirements I was able to make sure it was technically sound by not needing to write each value into each test. This also allowed for a quick update to the test in case there was a change in the requirements. Here is a sample of my code that showcases this:

**void** testContactIDWithMoreThanTenCharacters(String contactID, String firstName, String lastName, String phoneNumber, String address) {

Contact contact = **new** Contact(contactID, firstName, lastName, phoneNumber, address);

**if** (contact.getContactID().length() > 10) {

fail("Contact ID has more than 10 characters.");

}

}

@Test

@DisplayName("Contact First Name cannot have more than 10 characters")

**void** testContactFirstNameWithMoreThanTenCharacter(String contactID, String firstName, String lastName, String phoneNumber, String address) {

Contact contact = **new** Contact(contactID, firstName, lastName, phoneNumber, address);

**if** (contact.getfirstName().length() > 10) {

fail("First name has more than 10 characters.");

}

}

@Test

@DisplayName("Contact Last Name cannot have more than 10 characters")

**void** testContactLastNameWithMoreThanTenCharacters(String contactID, String firstName, String lastName, String phoneNumber, String address) {

Contact contact = **new** Contact(contactID, firstName, lastName, phoneNumber, address);

**if** (contact.getlastName().length() > 10) {

fail("Last name has more than 10 characters.");

}

}

I ensured that my code was efficient by continuously testing it. For this part, I will use address as an example. To check the address and make sure that it is not longer than 30 characters. This would be a repeated test that I would write in order to check for any addresses that do not meet the required length.

@Test

@DisplayName("Contact address cannot have more than 30 characters")

**void** testContactAddressWithMoreThanThirtyCharacters(String contactID, String firstName, String lastName, String phoneNumber, String address) {

Contact contact = **new** Contact(contactID, firstName, lastName, phoneNumber, address);

**if** (contact.getaddress().length() > 30) {

fail("Address has more than 30 characters.");

}

}

Reflections

**Testing Techniques**

Testing techniques that were used in this project are unit testing, boundary value analysis, equivalence partitioning, and decision tables.

**Unit testing** is a white box testing technique that involves testing individual units of code. This is the most basic form of testing and is essential for ensuring that the code is done correctly, and functions as expected.

**Boundary value** analysis is a black box testing technique that focuses on testing the boundaries of an input and output value. This helps in ensuring that the code can handle extreme values correctly.

**Equivalence partitioning** is a black box testing technique that involves dividing the input space into equivalence classes. This helps to ensure that the code can handle all possible inputs correctly.

**Decision tables** are a black box testing technique that uses a table to specify the possible combinations of inputs and outputs. This helps to ensure that the code handles all possible combinations of inputs correctly.

Other testing techniques that I did not use for this project were integration testing, system testing, and acceptance testing.

**Integration testing** is a black box testing technique that involves testing the interactions between different units of code. This is typically done after integration testing has been completed.

**System testing** is a black box testing technique that involves testing the interactions between different units of code. This is typically done after unit testing has been completed.

**Acceptance testing** is a black box testing technique that involves testing the system by users or customers. This is typically done after system testing has been completed.

The practical uses and implications for each of the testing techniques are as listed.

**Boundary value** analysis is a good choice for projects where the input and output values are critical. **Equivalence partitioning** is a good choice for projects where the input space is large or complex. **Decision tables** are a good choice for projects where there are a large number of possible combinations of inputs and outputs.

**Mindset**

The Mindset that I adopted when working on this project was one of trial, error, and precision. When looking at the assignment I knew that the test portion was going to be something that I would need to approach with caution. Complexity can be a large factor when doing testing, more complex code can lead to over-testing, or a test could introduce a bug unintentionally.

The simple nature of the code makes sure that the review is accurate and unbiased. More complex code could get in the way and cause bias within the review. This can lead to unintentionally covering up a flaw in the code due to the belief that tests cover a section of code that was overlooked.

Finally, it is important to be disciplined in your commitment to quality as a software engineering professional. This means not cutting corners when it comes to writing and testing code. You also need to be willing to invest time and effort into improving the quality of your work. This can be done by being disciplined in your commitment to quality, you can help to ensure that your software is reliable and meets the needs of your user.