Logan Martha

Professor Gudivada

CS – 320 Software Test, Automation QA

14 October 2024

1. **Summary**

For the module three milestone, I approached the assignment in a step-by-step approach. I took each specific task for the assignment and finished each task one by one. The two major requirements when creating this code were to ensure that the contact variables were never null and matched the desired lengths, meaning not too long. Utilzing condition statements that didn’t allow the contact variables to be null and exceed the desired length was what I used to complete this task. To test this, when creating the JUnit Tests, I made sure that the error message would appear if either of those conditions were not met.

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Contact(null, "Logan", "Martha", "7247994234", "6187 Brown Road");

});

The code provided is an example of how I tested the contactID variable within the Contact class. This specific code tests a new Contact input with a contactID of null, which is not allowed based off the requirements. If the code if done correctly this specific test case should pass, which it does. When it comes to the effectiveness of my code, the overall coverage percentage reflects just that. It may not be 100% coverage, but when you look back and see what is not covered, these lines of code do not need to be. The lines that are not covered are those of variables being changed only after a tested bit of code runs successfully. The red meaning, not covered in this example:

public void setFirstName(String setName) {

if (setName == null || setName.length() > 10) {

throw new IllegalArgumentException("Invalid First Name");

}

this.firstName = setName;

}

public void setLastName(String setName) {

if (setName == null || setName.length() > 10) {

throw new IllegalArgumentException("Invalid Last Name");

}

this.lastName = setName;

}

public void setPhone(String setPhone) {

if (setPhone == null || setPhone.length() > 10) {

throw new IllegalArgumentException("Invalid Phone Number");

}

this.phone = setPhone;

}

public void setAddress(String setAddress) {

if (setAddress == null || setAddress.length() > 30) {

throw new IllegalArgumentException("Invalid Address");

}

this.address = setAddress;

}

}

Also, the accessor methods are not tested, as they just allow the code to receive specific variables from the overall class to utilize them within the code. To ensure my code is technically sound, the use of comments was an emphasis for me in this project. Doing this allows me to go back and make quick changes if necessary. I will also be able to go back and use this assignment as a reference point in order to help with any future JUnit tests that I may need to recreate. This is an example of commented code I had within my assignment:

*@Test* // CONTACTID TEST

void testContactID() {

// TEST FOR NULL (THROWS ERROR IF THERE IS NULL)

When it comes to the efficiency of the code, object-oriented programming is one of the staples of that. Creating the code in a class format allows to keep the code clean and efficient. For example, providing accessor classes in order to minimize any redundant code.

// GETTERS

public String getContactID() {

return contactID;

}

public String getFirstName() {

return firstName;

}

public String getLastName() {

return lastName;

}

public String getPhone() {

return phone;

}

public String getAddress() {

return address;

}

1. **Reflection:**

When it came to the three milestones, we used the same acceptance testing techniques with JUnit testing. One technique I used was unit testing. This would ensure that each individual part of the code was functioning as intended. Some characteristics of JUnit testing is that they are automated. Since the test is integrated withing the coding environment, the test run and check automatically. Another great characteristic I love is the immediate feedback. Being able to see the results of your test immediately withing the IDE is nice. Another technique we used was JUnit Coverage. This allowed to see a measurement of the overall testing of the codebase. It would give a percentage of the entire code base of what was tested and what wasn’t. This was great to make sure all parts of the necessary code were tested. It would also go and highlight what exactly was tested.

One technique we did not use was integration testing. This is because there was no interaction between multiple components of the software. So far, all we have done was the backend development process. If we were to work with front and backend development, making sure we used integration testing to ensure that any extra interfaces such as APIs, databases, and external services were running along with the codebase correctly.

Between the two techniques of testing, integration and acceptance testing, they both have important roles within projects. Integration is critical when you are using multiple services that need to interact with each other within the software. This helps with interface interaction to ensure smoothness. It also helps with verification within external systems that need it such as databases. When it comes to acceptance testing, this form is crucial for client-driven testing. Making sure the components of the code match with the requirements of the customer are what this technique excels at. Overall, both techniques are beneficial and there are plenty more testing techniques that can be used.

When it comes to the mindset, I just wanted to provide the best quality code that matched the requirements of the customer. That was my only goal. I did not care if everything exactly aligned with my ideology, but I was determined to make sure the customer received what they wanted. When testing the code, it is important to appreciate the complexity of the testing. I feel that when you have to go back multiple times to ensure the testing is accurate and covers the majority if not all of the code, it proves that there is something meaningful going on in the program. For example, when trying to ensure over 80% coverage of the project, I had to go back and create more test cases to test the validity of a successful use of the program, rather than testing unsuccessful use of the program. When it came to any bias, I was not worried about this. My goal was to meet the requirements in any way possible. If this meant to scrap what I had and try another way, I would. Luckily, I did not have to do that, but just had to add from my initial tests. Lastly, when it comes to the discipline of truly testing the overall code is super important. After researching and reading about the errors that can cause detrimental damage to those who use a program, it really opened my eyes to the importance of efficient testing. The program we were assigned did not necessarily deal with precious data, but in the future, if I were assigned to a project that requires precious data, the testing process would need to be lengthy and excessive to ensure safety within the program. Cutting corners in code testing is bad for everyone involved and must be eliminated and avoided at all costs