Jaymie Johnston

CS-320

Professor Jeff Phillips

April 19, 2024

7-2 Project Two

In project one, we complied code for three features of a mobile application. The features were Contact Service, Task Service, and Appointment Service. In addition to these features, we implemented JUnit tests to test the various requirements of each feature. For example, a requirement of the Contact Service was that the address field could not be longer than 30 characters and could not be null. This is illustrated by the following code:

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

throw new IllegalArgumentException("Invalid Address. Cannot be longer than 30 characters and cannot be null.");

}

The JUnit test for this requirement was:

//Test if Address is too long

@Test

void testContactClassAddressTooLong() {

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

new Contact("ContactID", "Tina", "Belcher", "5559894128", "AddressIsWayTooLongNowAndNeedsToBeShorter");

}); }

//Test if Address is Null

@Test

void testContactClassAddressNull() {

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

new Contact("ContactID", "Tina", "Belcher", "5559894128", null);

}); }

This JUnit test was testing an address that contained more than 30 characters, as well as another test to see if the address was null. Additional requirements were tested in the other services in a similar manner. All the requirements in each of the features had corresponding JUnit tests. The overall quality of my JUnit tests was moderate. I was able to achieve 80% coverage or above on several of the tests, while others fell a bit short of that threshold, as highlighted in the below screenshots.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

To ensure that my code was technically sound, I referred to the resources that were made available. I made sure to run the code every few lines to be sure to catch any errors early on. Even a simple error, such a missing bracket, can cause multiple errors in different lines of code. It is important to be continuously testing the code as it is being written. After every block of code was written for the JUnit tests, I could run the test to make sure that it was composed properly and that it returned the desired results (no errors). If errors were found, I attempted to fix them mostly with trial and error. I did refer to the resources as needed, as well as found some useful tutorials online. I also tried to include code that was appropriate for the guidelines and requirements to make sure my code was efficient. For example, in Task Services, I made sure that Task class objects could be created:

@Test

void testTaskClass() {

Task taskClass = new Task("FirstID", "Linda", "TallRedColdSunny");

assertTrue(taskClass.getTaskID().equals("FirstID"));

assertTrue(taskClass.getName().equals("Linda"));

assertTrue(taskClass.getDescription().equals("TallRedColdSunny"));

}

The software testing techniques used for this project was mainly JUnit testing. JUnit testing falls under the white box testing category as it tests individual components of a code. This is shown by the various JUnit tests that were created to test each of the requirements of the three different features. There are many different types of software testing techniques that were not used in this project. An example of one that was not used is exploratory testing, where no specific plan is used for testing and the tester uses their experience and knowledge to search for any defects in the code. Another testing technique not used would be security testing. For this project there weren’t any security requirements outlined, so this was not a technique that would have been used.

Exploratory testing can be used in a situation where the tester has a lot of knowledge of what to look for in terms of defects. This is essentially an attempt to “break” the system. Security testing is important in situations where there are specific security requirements that the code must meet. This would be vital in programs that contain sensitive information for business and/or individuals. For example, a bank or financial institution would require security testing.

The mindset that I had while working on this project is that of a student who still has a lot to learn. I was cautious and aware of the code I was compiling and how it related to the requirements for the project. I tried to keep things simple and use the resources I had available to me. I worked to fix any errors with trial and error and to go over my code multiple times looking for any possible areas of concern. It was important for me to remember that the codes worked together and that an error in one area could also cause an error in another area. For example, failing to set the getters in the classes would result in errors in other areas of the program, including the JUnit tests.

To limit bias, I tried to look at things with an open mind and the understanding that I am still learning and therefore will not be perfect. Bias can come into play when testing your own code if you are not open to the fact that you will make mistakes. I can also imagine that it would be possible for a software developer to unethically manipulate tests or pieces of code to give a desired outcome. Taking shortcuts or using bad practices to get a specific outcome could have serious consequences down the line. Subpar code could negatively impact so many aspects of our society. We, as a society, have high expectations that places like a bank, or a hospital will keep our information safe. Or that the technology that is used in our modes of transportation is sound and won’t contain bugs that could cause us serious harm. Holding ourselves to a high standard of integrity will only make us stronger software developers. Knowing that the product we create is made with integrity and is of high quality will benefit everyone involved.