Project Two

Jeremia Faust

Professor Bryant Moscon

June 20,2021

**Testing Approach**

All requirements for Contact.java, ContactService.java, Task.java, TaskService.java, appointment.java, and AppointmentService.java were very similar. This made my approach to testing very similar, thus I only had to change a few things to adapt the tests to the different objects and services. For the objects we were given a video that walked us through a good way to setup by setting up the outerbounds and throwing exceptions when the data is out of bound. This allowed the tests to catch the exceptions for the test using this code:

void testPhoneNumberToLong() {

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

new Contact("1234","1234","1234","12345678911","1234");});

}

For the ContactService I knew from my last class that, an array list would work great for this project. Unfortunately, I could not get the exceptions to work like in the objects. This is where I got help because I was stuck. It was then recommended to use Boolean statements that output true or false. This would allow me to use assertEquals for the test as such:

@Test

void testAddContactfail() {

ContactService contactService = new ContactService();

Contact contactOne = new Contact ("1111","1234","1234","1234567891","1234");

assertEquals(true, contactService.addContact(contactOne));

assertEquals(false, contactService.addContact(contactOne));

}

This test uses the output from the Boolean statements and compares to the current command assertEquals(true or false, …….). I have found I like using Boolean for testing because of the yes or no nature of them. It makes it very easy to understand and to find problems.

**Quality**

For all my objects and my services, I was able to reach hundred percent. Originally my first attempt was like 50% percent so when I started the second milestone, I realized my mistake very quickly. My first problem in the services in the addContact() there is a statement that checks if the contact already exists. For some reason it was not working at all. I spent hours trying to make it work, finally I decided to get help and here I needed to add an override to the object to make it work.

@Override

public boolean equals(Object object) {

if (this == object) {

}

return true;

}

Once this was added everything started to work. Another problem was my tests were not written correctly and I was writing it in a way where the test was giving me false positives. The only problem I am continuing to have is in the objects tests the assertThrows the red part is showing up as not used. I also noticed this in the video, but it is throwing off the overall score for the test coverage.

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

**new** Contact("1234","1234","1234",**null**,"1234");});

}

**Technically Sound Code and Efficient Code**

This code is both efficient technically sound and efficient, as we hit 100% coverage on all the code files, Contact.java, ContactService.java, Task.java, TaskService.java, appointment.java, and AppointmentService.java. At no point there is no code being unused even though the assertThrows is saying unused code is there. Both the services have the same warnings this is because one an entry is found it will not continue looking through the array.

**boolean** existingContact = **false**;

**for** (Contact list : contacts) {

**if** (list.equals(contact)) {

existingContact = **true**;

If you look at the tests for all the tests are very short and quick. All the tests for tasks took a total of 0.246 seconds to complete. This shows that there was nothing that requires a lot of time so it will not use a lot of memory or time to complete. This is most important when you are testing large projects. The old saying “time is money” is very applicable here. All this code is reusable as we can see between Contact.java, ContactService.java, Task.java, TaskService.java, appointment.java, and AppointmentService.java. I was able to adapt the code very easily with few changes. This saves on time creating tests when you can just reuse them. When you compare difference between addContactFail() and addTaskFail() there is very little difference, but it does a good job testing the code. All we have to do is change a few things to match the objects besides the name changes you can see the differences of the object sizes and the different required strings sizes.

@Test

**void** testAddContactfail() {

ContactService contactService = **new** ContactService();

**Testing Techniques**

The type of tests that are used for this project is White-Box testing. In white box testing the tester is also a programmer and is more focused on the code itself. We are also trying to test every pathway to get as close to 100% coverage as possible. We are doing a form a unit testing which is generally done at a lower level which is part of white-box testing.

Other types of testing would be black-box testing and Ad-hoc testing. Black-box testing is testing where the tester has no knowledge of the code or the internal structure of the program. The tests tend to be done from the users point-a-view. The tests are generally created using the requirements to test the functionality of the program. A lot of times these tests are created before the code has been written. Ad-hoc testing is a more informal type of testing where the tester does max/min testing as the user to try to break the program. The tester then records how they found the bug so that it can be recreated. This allows the developers to narrow down the problem in the code to fix it.

White-box testing tend to be very complex and expensive. It needs to be tested as code is being written then again after the code has been written, then again after any modifications has been made. It allows for most of the code to be tested and shows where there are problems and where there could be problems. Black-Box testing all that is important is understanding the user’s point of view. The tests take in the input and the program spits out the output. This type of testing is done early after the code has been written but can cause problems if the user interface changes. Another problem is that the tests may not get full coverage of the entire program. Ad-hoc testing tend to be used after code has been written and can test small part to the entire program. It is not as through as of testing techniques but it also the cheapest. I see ad-hoc testing as a form of a beta test as we are trying to stress the system to provide errors. All three of these tests are important and when used together you end up with a nearly bug free system.

Contact contactOne = **new** Contact ("1111","1234","1234","1234567891", "1234");

*assertEquals*(**true**,contactService.addContact(contactOne));

*assertEquals*(**false**,contactService.addContact(contactOne));

}

**VS.**

@Test

**void** testAddTaskFail() {

TaskService taskService = **new** TaskService();

Task taskOne = **new** Task ("12341","1324","123456");

*assertEquals*(**true**,taskService.addtask(taskOne));

*assertEquals*(**false**,taskService.addtask(taskOne));

}

**Mindset**

When creating the code and the test I had to take the mindset of a programmer and a tester. As a software programmer I had to be very careful when creating the code and the tests that go with them. Part of that is my experience is very little, so I was forming the code around the tests. I had to be careful not to code my test just to get a passing test. One thing I noticed between the objects and the service is how they interact between each other and the tests. The objects take the data and checks to make sure the data is within bounds and returns it. The services manipulate the data and then then stores it in an array list. The services also return true or false pending if the data successfully manipulated. The tests introduce the data and pass-fail tests are performed. When looking at the services tests you can see both the services and the objects working together especially when the data is entered wrong you can see the exceptions from the objects, or you will get an error say this should say true but it is false.

When the developer does both the programming and the testing it is very difficult to limit bias because the developer tends to become invested in the code. This makes it very easy to write the tests in a way that force a false positive. I noticed that I was doing just that but, in my case, I was learning and as I learned and practiced, I could recognize where I provided false positive tests and was able to fix them. Most of the time I had to go back to the object and the services and changed the code. An example of a false positive was when I wrote the tests to provide all passes when I went back to it I realized that there was a problem that I needed to fix. In this case I assertFalse and the program should be outputting true but was outputting false, thus creating a false positive.

As a software engineering professional, you should never cut corners when writing code or in testing because it could end up with a bad program that cost the company lots of money to cost people lives once it releases. A great example of a bad program is a game called cyberpunk 2077. The developer was forced to cut corners because of management made promises to release at a certain date and would not listen to its creative teams. The result was a buggy game to the point of being unplayable. It hurt the company’s reputation and cost them millions of dollars. Sony even pulled the game from their online PlayStation store. Almost a year later they are still fixing the problems and still have not been allowed back on the PlayStation store . To limit technical debt, we have to follow good coding practices, follow good testing practices, and create plans and track results. It is very easy to cut corners to fulfill a timeline or to stay within a certain budget, but it is important you do not fall in those traps easily and without thought. In the case of the cyberpunk game, it would have been cheaper to extend the timeline than to release a buggy game.