SNHU

Project Two

Jonathan Sussan

Kalysa Wilson

### CS 320 Software Test Automation

12/08/2021

**Summary**

**a.** My testing approach aligned with the software requirements as in both the Contact Service and Task Service assignments, I was able to successfully test my code with no errors. By using in-memory data structures, I was able to verify the Contact Service and Task Service worked when trying to add, update, or delete information. An example of this is in the code below, where I tested to make sure that if the Task ID was character length was greater than 10 or was null, the code would throw a new illegal argument exception:

// test task id, 2 tests

@Test

**void** testTaskIdLength() {

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

**new** Task("801312801312", "Peter", "Is Spiderman");

});

}

@Test

**void** testTaskIdNull() {

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

**new** Task(**null**, "Peter", "Is Spiderman");

});

}

The overall quality of my Junit tests for Contact Service and Task Service is very highly effective. Although no code is perfect, and in essence cannot reach 100% coverage, I feel I was able to get to 95% coverage. I was able to test and make sure that exceptions would be thrown anytime a condition that was in the requirements was met, such as character length or null values. It helps to keep a checklist and be organized to make sure I was able to run effective tests.

**b.** I ensured that my code was technically sound by keeping my code organized and uniform, along with providing good comments. I made sure to keep both the Contact Service and Task Service uniform and in line. An example of my code being technically sound is below:

// constructors

**public** Task (String taskId, String name, String info) {

// validate constructors

// task id may not be null or have more than 10 characters

**if** (taskId == **null** || taskId.length() > 10) {

**throw** **new** IllegalArgumentException("Error, please enter a valid task ID.");

}

// name may not be null or have more than 20 characters

**if** (name == **null** || name.length() > 20) {

**throw** **new** IllegalArgumentException("Error, please enter a valid name.");

}

// info may not be null or have more than 50 characters

**if** (info == **null** || info.length() > 50) {

**throw** **new** IllegalArgumentException("Error, please enter valid information.");

}

My code was efficient because I only included code that was needed for the program to function correctly and kept my code well organized. It was written as simple as possible and didn’t complicate anything. Constructors, accessors, mutators, etc. were all nicely grouped together and well commented. The code below shows how in my Contact test, I was able to efficiently test for phone number length being too long or too short, null, and contain only digits:

// test phone number, 4 tests

@Test

**void** testPhoneNumLong() {

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

**new** Contact("801312", "Peter", "Parker", "818555131313", "Queens, New York");

});

}

@Test

**void** testPhoneNumShort() {

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

**new** Contact("801312", "Peter", "Parker", "81855513", "Queens, New York");

});

}

@Test

**void** testPhoneNumNull() {

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

**new** Contact("801312", "Peter", "Parker", **null**, "Queens, New York");

});

}

@Test

**void** testPhoneNumDigits() {

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

**new** Contact("801312", "Peter", "Parker", "818555131z", "Queens, New York");

});

}

**Reflection**

**a.** The software testing techniques I employed for the milestones are part of the JUnit testing framework. A Junit test is a method in a class that is only used for testing, which is the test class. To mark a method as a test method, I first needed to annotate it with the *@Test* annotation. This allows for the method to execute my code under test. Throughout the three milestones, I used assert statements to check an expected result versus the actual one. If the actual result does not match my intended one, a message would show me there was an error. This is especially helpful in identifying and fixing problems. Below is an example, taken from my *AppointmentServiceTest.java* file:

// test task and task service object

@Test

**void** testAppointmentService() {

AppointmentService appointmentService = **new** AppointmentService();

Appointment appointment = **new** Appointment("801312", **new** Date(), "Spiderman is booked");

*assertFalse*(appointmentService == **null**);

*assertFalse*(appointment == **null**);

}

You can see in the example above, after initiating the test method and creating a new appointment service, I used assert statements to show false if there is a null value. The assert statements I used most frequently in these milestones were *assertEquals*, *assertThrows*, *assertTrue*, and *assertFalse*.

Other software testing techniques I could have used are *assertAll*, *assertNotNull* and *assertNull* to name a few. The statement *assertAll* is used to check that all asserts are checked because if an assert fails a test, JUnit will stop executing the test and not all asserts will be checked. The *assertNotNull* and *assertNull* are straight forward in checking if a value is null or not. Since I am new to JUnit testing, I wanted to keep it simple and mainly use *assertTrue* and *assertFalse* statements so that I become comfortable using these methods.

Testing those certain exceptions are thrown by using assert statements is a very helpful tool. This is done by defining the expected exception class and providing code that should throw the exception. I mainly used *assertThrows* exceptions to show if an *assertTrue* or *assertFalse* statement triggered. These statements checked to see if the expected result is the actual one. I know we are just scratching the surface in terms of JUnit testing, but I’m fascinated to learn more about this subject throughout the rest of this course.

**b.** The mindset I adopted while working while working on this project was to be cautious and not overcomplicate anything. While writing code, I was constantly thinking if I would be able to adequately test the code. Even though I understand that I am not overly experienced as a developer, I do understand that when writing code, it is important for testing purposes to write code that is simple for the developer to test. Using this mentality, I was able to very easily test my code to make sure it runs as intended. I could of wrote code that was more complicated, but I was not comfortable enough that I was able to adequately test it.

I was able to limit bias during the review of my code because I or another developer would have been able to write it in a more advanced fashion. However, I wrote the code as simple as possible to my understanding level, so that I would be able to easily test it and be confident in my testing methods. For example, during my JUnit tests I could have used *assertAll* to check that all asserts pass the test and fail if one does not. However, I felt more comfortable performing each test separately to simplify them for my experience level. Like many things, I believe that bias plays a role when writing code. It’s easy to make mistakes with JUnit tests when applying them to your own code. For example, you may have used an *assertFalse* test incorrectly where an *assertTrue* was needed.

It is important to be disciplined in your commitment to quality as a software engineering professional because the code we write, can cause harm in several ways out in the real world. For example, vulnerabilities in code in a financial application may lead to a person’s sensitive data be stolen which can cause fraud. Coming into this course, I didn’t know what to expect. How to incorporate JUnit testing in my code to increase the quality of it was an important lesson I learned. Going forward, I now have a better understanding that I should incorporate testing into all code that I write. While this isn’t a 100% full proof method to ensure our code meets quality standards, it is a great tool to have in our arsenal.

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved from  
<https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb>