**Summary**

In writing the tests for the application that has been assigned, I made an effort to test each requirement of the classes specifically. I tested each attribute thoroughly by testing each set method for the attributes to ensure that they adhered to the requirements and also tested the same attributes when adding a new task or contact. For instance, the requirements for the task Name attribute are that it can’t be null or longer than twenty characters. To test this, I have a test called testSetNameTooLong that tests when a task is created and the user attempts to manually set the name longer than twenty characters. I also have another test called testTaskNameTooLong that tests when a task is created that initially has a name that is longer than 20 characters. In a similar fashion, I have testSetNameNull and testTaskNameNull, which both test that an exception occurs when the name is null.

While working on the Task and TaskService tests, I found the option to view the coverage of my tests. I did notice that my testing coverage wasn’t as high as I had expected initially but after making some changes, as well as learning that I should look at the coverage of my application and not my tests, I now have 100% coverage for my application files. This tells me that every line of code in my application Java files are covered in a test and that the tests that I’ve written are effective in testing my product.

I ensured that my code is technically sound by making the tests as static as possible. When I made a change, I had the tests check for the exact output that is expected so that I didn’t depend on another function to confirm the tests. In many of the tests, I checked the output of the data prior to performing the test and then confirmed again after the test was run. For instance, in the testDeleteContact test, I confirmed that the list of contacts included two contacts as I added them.

**if** (target.getId() == "1") {

*assertTrue*(target.getFirstName().equals("Greg"));

*assertTrue*(target.getLastName().equals("Roberge"));

*assertTrue*(target.getPhone().equals("6031234569"));

*assertTrue*(target.getAddress().equals("20 Union St"));

}

**if** (target.getId() == "2") {

*assertTrue*(target.getFirstName().equals("John"));

*assertTrue*(target.getLastName().equals("Smith"));

*assertTrue*(target.getPhone().equals("6034567891"));

*assertTrue*(target.getAddress().equals("35 Cilley Rd"));

}

I then created a new list called ‘expected’ which only had the second contact. After I ran the deleteContact command to remove the first contact, I used the ‘expected’ list to confirm that only the second one was remaining in the list.

List<Contact> expected = Arrays.*asList*(contact2);

service.deleteContact(contactList, "1");

*assertThat*(contactList, *is*(expected));

This method of testing took as much of the computer error out of the equation and confirmed exactly what I was looking for.

I ensured that my tests were efficient by using BeforeEach and AfterEach to set up the contact or task list in the service test files. I created any global variables at the start of the tests so I didn’t have to spend processing time creating the same variables multiple times.

**private** List<Contact> contactList = **new** ArrayList<Contact>();

**private** ContactService service = **new** ContactService();

@BeforeEach

**void** setupList() {

Contact contact = **new** Contact("1","Greg","Roberge","6031234569","20 Union St");

contactList = **new** ArrayList<>(Arrays.*asList*(contact));

}

@AfterEach

**void** teardownList() {

contactList.clear();

}

I also didn’t do any additional computing during the tests unless it was necessary. Each test was configured to test one requirement so that if any of the tests fail, it’s easier for me to figure out where the failure is and what part of my code needs to be addressed.

@Test

**void** testContactAddressTooLong() {

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

**new** Contact("123456789","Greg","Roberge","6031234569","20 Union St Merrimack New Hamps");

});

}

**Reflection**

In the testing that I’ve done for the milestones, I’ve done unit testing to test each piece of source code to verify that they are working as expected. This included the setup, exercise, verify and teardown states. I used the @BeforeEach and @AfterEach Junit annotations to assist in the setup and teardown actions. The techniques that I used were mainly examples of white-box testing although I did perform some black-box testing also. For the white-box testing examples, I used fault injection which was performed by adding data values to my objects that went against the requirements to confirm that these values resulted in assertions as expected. I also used mutation testing to test the changing of values of an existing object (Garcia, 2017). As far the black-box testing, I used systematic testing which was utilized to confirm that the requirements were met and that the functionality of adding, updating and creating the objects was successful. These tests allowed me to achieve 100% code coverage for testing the application.

Some of the testing techniques that I didn’t use were many of the black-box techniques as those are normally used when the internal code and logic of the system isn’t known. These include random testing, GUI testing and model-based testing (Garcia, 2017). Because I had written all of the code, I was able to test it using specific functions and settings. The testing requirements and failure scenarios for the applications were pretty straight-forward so the random testing of changing values was not necessary and I had tested using fault injection to catch the failure scenarios. Since there is no graphical user interface that has been developed for this application, the GUI testing was also not necessary. As I stated earlier, because I know the internals of this system, I also didn’t need to perform any model-based testing as I have deeper knowledge of the product beyond a model or description level.

Many of the black-box testing techniques would be good to use if I wasn’t a developer of the product. For instance, if I was a member of the Quality Assurance team that only had the requirements or the description of how a product is meant to work, then many of these would be useful. Using models that describe how a feature is supposed to work would be good to be able to test each individual feature based on the requirements and expectations of the user. GUI testing can also be useful to try and give the user the best experience when they are using your product. Many times, using the product on a different operating system can affect the way that it works so running these tests on different systems and versions is helpful. Random testing can also be used for a larger project when the time is put into trying to break a system. Basic testing functionality run by the developer can be used because they know how a program is going to break but random testing can be used to try and find ways to break it that aren’t yet known to the developer. There are many different ways to ensure that a program works as it should and using many of these techniques together can give a full complement of tests to flush out as many of the possible bugs prior to release.

While working on this project, the mindset that I originally had was to test all of the main functions to ensure that they worked as they should. After I realized that I was missing many necessary tests, I took a step back and looked at each individual step of the project. I started to implement tests of any function or method that could change the data for an object or the object itself. It was imperative that I understand what these functions and relationships are and how they interact with each other to be able to have complete testing of the application.

For instance, there are four ways that an ID for an appointment can be set. These include setting the id when creating the appointment, changing it after the appointment is created and using the appointment service to add or update the appointment. To test each step of these methods, I have two tests for the setId method and two for when an appointment is created, one for each requirement of it not being null and not being longer than ten characters. I then have additional tests for adding an appointment using the service and updating it using the service. Knowing that all of the services are related to the methods of the object, helps me to fully test the scenarios without creating additional tests to confirm the same requirements with the appointment service.

I tried to limit my bias in reviewing the code by doing my best to not assume that any part of it works without walking through it. As I stated above, I tested each method that was able to edit or set a part of the object. I also read that data back to make sure that the output was as expected. I could imagine that bias could creep into my tests if I was testing my own code as it would be very easy to skip writing a test of a basic method if I was in a time crunch or to skim over part of my code because I’ve written it many times before. In many professions or sports, it’s very simple to forget the basics and only concentrate on a new skill or characteristic but in software development, all of the basics must also be tested. Having separate testers and developers is a great way to avoid assumptions of talent or skill as the testers are more apt to try and break the software while, as the developer, I may be more interested in proving it works.

Being disciplined in my commitment to quality as a software engineer is important because sometimes the smallest mistake or missed test can have a large negative effect on a product. The code that I write today will have other code added to it in the future so making sure that my code can be trusted to be used is imperative so that others can add or improve functionality without necessarily having to worry about whether my initial code works. The more exhaustive that testing is done, the better the product that will be released and the less time that will be spent fixing bugs rather than adding new functionality. I hope to avoid technical debt by taking the time to understand the code that I’m writing and the effects that it may have on code that is already there. Writing unit tests to confirm my functionality and avoid regressions as much as possible will help as well as explaining the interactions between different areas of the code to assist QA in hardening the tests to find any flaws before it is merged to production and released will also be helpful.

**Resources**

García, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing