**To what extent was your testing approach aligned to the software requirements?**

The testing I did on both the Contact/Contact service and Task/task service programs were directly related to the requirements given. The requirements acted as a blueprint for testing, for each requirement there was a corresponding test. For example, the Contact class required that a contact ID was not empty and that it was not longer than 10 characters. After coding this requirement out I made test cases for each piece of the requirement, so I tested that the contact ID wasn’t empty and that it wasn’t longer than 10 characters. I did this for every requirement in each program. There were some tests that did not directly correlate to the requirements, for example , I tested that the Contact class was actually working and that Contacts were actually put into a Contact list (in the Contact Service class). The reason these tests were preformed is because we want to make sure each part of our code was working.

**Defend the overall quality of your JUnit tests for the contact service and task service.** **In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?**

By running a Junit coverage test I was able to visually see how much of the program was tested. The coverage test also highlighted the portions of code that were fully tested and which ones were not, aiding in the completion of testing. Since I knew that JUnit testing aims to test each individual block of code, I knew before preforming the coverage test that there was effective testing because I ensured that I hit every block of code with a test.

**How did you** **ensure that your code was technically sound?**

I ensured my code was technically sound by using JUnit annotations like @Test in order to test the code and @DisplayName in order to have a label the test reports which allowed me to see which tests were being run and which ones fail. Additionally, I used various assertion methods and lambda expressions in each test which carried out multiple different tests. In the Contact.java program there were illegal argument exceptions thrown if the conditions were not met. When I tested that in the ContactTest.java program I was able to use a cool assertion method that contains a lambda expression to ensure than an argument is thrown

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

**new** Contact("1234", **null**, "McGuire", "2063758969", "15 seattle st");

});

**How did you ensure that your code was efficient?**

One way I ensured that my code was efficient is by using logical operators when I was creating the conditional loops in both the Contact.java and Task.java. I was definetly more efficient in the TaskTest.java program than the ContactTest.java because the TaskTest was able to do similar tests with less lines of code. For example, in ContactTest.java I used two separate tests to see if a name was null and if a name was too long.

//testing that an exception is thrown when First Name is too long

@Test

@DisplayName("First Name too long exception")

**void** testLongFirstfield() {

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

**new** Contact("1234", "Raecheliscool", "McGuire", "2063758969", "15 seattle st");

});

}

//testing that an exception is thrown when First Name is null

@Test

@DisplayName("First Name null exception")

**void** testNullFirstfield() {

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

**new** Contact("1234", **null**, "McGuire", "2063758969", "15 seattle st");

});

}

Since these were separate tests I had to write more code than I would have if I just put them both under the same test. I was more efficient in the TaskTest.java program by doing similar tests in less code

//testing that the Name throws an exception if its null or greater than 20

@Test

@DisplayName("name incorrect length test")

**void** testName() {

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

**new** Task("1234", "RaechelElisabethMcgui", "clean the floor");

});

Assert.*assertThrows*(NullPointerException.**class**, ()->{

**new** Task("1234", **null**, "clean the floor");

});

}

**What were the software testing techniques that you employed for each of the milestones?**

For each milestone I employed the white box testing technique. White box testing implies that the code is both visible and understood by the tester, revolving around internal testing of the program or system. White box testing verifies the flow of the program by testing a series of predefined inputs against expected outputs, so that when an unexpected output appears, we know we have a bug. The type of white box testing used was unit testing, which is usually done by the developer concurrently with programming. Unit testing tests each individual unit or function in a program to ensure that they’re working as expected. Finally, for each milestone I concluded with a coverage test, which allows us to see what sections of code the unit tests hit and which ones it didn’t. Preforming coverage tests gives us a good idea on how much of the code was tested (the more coverage the better), which parts of the code were tested , and which were not, that way if we need to create more unit tests we can.

**What are the other software testing techniques that you did not use for the milestones?**

Another software testing technique that was not used in these milestones was black box testing. Black box testing implies that the tester does NOT have knowledge of the programs inner workings and is only concerned with the end user experience. Testing using the black box technique verifies that the program is working as expected by using test cases derived from requirements. The tester can test those cases through a user end point of view in order to verify that the expected outputs are generated per each input, if not than a bug has occurred.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

For each software development project, we need to both verify the system and validate the system (V&V) to ensure were delivering the right product that works correctly. Both black and white box testing techniques are a way to validate that the product is working as expected by testing and executing the program. White box testing is conducted at lower levels including unit testing and integration testing, these tests are usually done early on, typically by the developer themselves. Black box testing is preformed later on in the SDLC after the code is written, including exploratory and performance testing, this is done through an end user perspective (through some user interface or HTTP endpoint). White box testing allows early detection of bugs while black box may not find them until later however, using both ensures that bugs can be caught on all different levels and that nothing “broke” throughout the project’s lifespan. Using white box testing allows to validate the inner workings of a program while black box allows us to validate the outer workings of a program (or the final system).

**Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing?**

Throughout testing the components of the project using JUnit tests I had to change my mindset from developer to tester. Throughout I was cautious to test each section of code, some needing multiple tests. For example, in the Contact class the first name couldn’t be empty or greater than 10, although those conditions were in the same function, I needed to create two tests to ensure that the length was correct. Each section of the project had two parts, for example there was a Contact class and a Contact service class that used the Contact class. Its important to ensure that the two can work both independently and together to create the desired result.

**Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?**

The biggest bias I think anyone can have while writing a program is overconfidence in the code they’ve written. When I was writing my program, I knew that certain parts would work, but that doesn’t mean I shouldn’t test it. Not testing everything as extensively as possible can lead to all sorts of bugs which is why its important to ditch the overconfidence. The job of a tester is to break the code, while the developer’s job is to create working code. If a developer has biases around their code, then it’s easy to be a lousy tester since, of course a developer doesn’t want to break their code or prove that it doesn’t work.

**Evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field?**

For any field, it’s incredibly important to provide quality products to your clients\costumers, which includes not cutting corners. Imagine buying a car, the manufacturers are professionals, and we assume they know what they are doing and taking their craft seriously, after all, we are paying a hefty price for the car. When you get the car, you notice the manufacturers cut some corners, maybe the fabric on the seats isn’t stitched well and the glove box is missing a screw. These problems may seem small but later your seat starts ripping at the seams and your glove box hinge completely breaks, now you have a big problem and you’re an angry customer.

This scenario can happen in software too, cutting corners and providing programs that aren’t of quality can cause problems, even if its later, causing you to have an unhappy client, who will have unhappy users, causing both of you to lose business and money. This is just one of many problems, bad software can lead to many things going wrong, sometimes bad software can even lead to catastrophes or worst, fatalities. Whether you’re working on a simple login page or a plane, its important to remember that we are professionals being paid to do this work. Ask yourself, would you fly on the plane that you just programmed?