## Summary and Reflections Report

In developing this back end of this application which allowed for contact, task, and appointment services for the user, I practiced and honed my skills in the area of unit testing.

The software requirements only directly specified the requirements of the main class files, with the contents of the test files needing to have been extrapolated off of these requirements as well as the principals and guidelines for creating unit tests.

When it came to my task service, contact service, and appointmentservice files, these were extremely closely aligned with the requirements. I included a constructor with appropriate parameters, some input validation for the null edge cases, as well as getters and setters for the contact, task, and appointment classes.

I included modifications methods for the array lists of objects for all of the service classes, such as add, delete, and update. I did these specifically depending on the exact requirements outlined for me.

I chose unit tests based more so on general principals.

A unit is an arbitrary separation of components. It is up to the code tester to use a consistent and beneficial framework of where to draw the line of one component/unit from another.

Unit tests are extremely important, so that when the code is having syntactical or logical errors, we can see the exact component where the problem is coming from. It is for this reason that is important to examine components as isolated cases, which can be tested for functionality by themselves.

However, if we break the code down into too many components, we are wasting time. For example, if we were to write a unit test for every getter and every setter in a codebase, this would be unrealistic. Since getters and setters are streamlined, it is logically not possible for a getter coded one way to be correct while a getter coded the exact same way is incorrect.

I decided to base the component division on the unique methods and constructors of the codebase that went beyond a generic framework. This meant, the edge cases checking for too many characters, or null amount of characters, as well as some of the constructor object instantiations, as well as most of the methods from the “service” classes such as adding, updating, and deleting objects from the array.

I know my Junit tests were affective based on coverage percentage, because my end project had many tests that passed as “runs” and few tests that were categorized as “errors” or “failures.” I also based the syntax structure of my Junit tests off of the readings and other resources provided, including the code outline provided to us in course announcements. I used this type of reasoning to decipher when to use a fail statement and when to use an assertion.

I ensure my code was technically sound by following the correct junit testing code syntax. For example, in ContactTest.java, I wrote:

*@Test*

*@DisplayName*("Contact First Name cannot have more than 10 characters")

void testContactFirstNameWithMoreThanTenCharacters() {

Contact contact = new Contact("OllyOllyOxenFree","LastName","PhoneNumbr","Address");

if(contact.getFirstName().length() > 10) {

*fail*("First Name has more than 10 characters.");

}

}

The fail statement prints if the test condition, (first name with more than ten characters) is met. This follows proper junit testing logic and syntax.

In ContactServiceTest.java, I wrote:

*@Test*

*@DisplayName*("Test to Update First Name.")

*@Order*(1)

void testUpdateFirstName() {

ContactService service = new ContactService();

service.addContact("Dr.", "Cross", "5555551111", "123 Lollypop Lane");

service.updateFirstName("Sven", "0");

service.displayContactList();

*assertEquals*("Sven",service.getContact("0").getFirstName(), "First name was not updated.");

}

The syntax structure here is slightly different, with the condition of the test being referenced in the name. But now the condition of the test is not a binary yes or no question. It is instead an action/process. For this reason, we use an assertion to check if said action/process has been successfully performed. We use assertEquals, comparing our updated first name, “Sven,” to the output of the method getFirstName(). If the assertEquals statement succeeds, nothing happens and the test completes. If assertEquals fails, the fail statement prints.

The code excludes what would have made it inefficient. We did hot conduct Junit tests for every getter, setter, arrray declaration etc. The reason for this is that these segments of code were quite streamlined and therefore unlikely to result in errors.

For example, the line....

ArrayList<Contact> contactList = new ArrayList<Contact>();

....might look complicated, but it is simply a generic array declaration, with no customized syntax or processes associated. For this reason, it would be inefficient to write unit tests for all three instances of arrayList declarations in the service classes.

Some software testing techniques that I employed in this project were dynamic testing in the form of junit based test files, which give feedback when the specific code segment is run, as well as unit tests since these are a type of unit test. I got to learn that a static vs dynamic test is an objective measure, but a unit vs system or integration test is something that the tester can arbitrarily define with his or her own boundaries, depending on what suits the project. I originally thought that Junit tests were a type of static test, since the main code files do not get run, but since the test files do get run, they are technically a form of dynamic test.

I did not employ any sort of integration or system testing. I know that integration uses the arbitrarily defined boundaries of the integration or two or more components, and tests these as one cohesive system. I do not know the specific techniques or software's involved in this but I look forward to learning in the future.

For every software development project or situation, unit testing/component testing will always be relevant. However, the cutoff where one defines individual units or components may change with the scale of the project, with bigger projects having larger components. These types of projects may rely more on integration and system testing due to their scale.

Duifferent software development methodologies can also effect the chosen overarching testing technique that should be in place the most. Software development methodologies include those such as agile, scrum, waterfall, etc.

The mindset I adopted for this project is an interesting thing to reflect on. I think at first I was just trying to absorb all of the new syntax as well as new concepts at the same time, but it wasn't until the second of third milestone that I really put these together in my brain as a new solidified concept. As the project came together I definitely appreciated how the different base classes and service classes came together to illustrate the complex backend of an application along with the test classes to check their validity.

It is important to be mindful of any potential bias during a small scale or large scale code review, which is a static testing method which can inform how dynamic unit testes will be set up. If one is testing their own code, they might be biased and not see things that might be obvious for someone with fresh eyes. It is for this reason as well as code testers having a refined specialty in their area which we often have a separate tester on a development team. This project made me think about discipline during commencement and moving forward. Cutting corners, turning devsecops to just devops, or being too arrogant to test one's own code, can be the loose end which ends up in glitches in the service after deployment, or a large-scale cybersecurity issues.

Well I was still in school I had many lessons on the importance of avoiding technical debt by always keeping these ideals of precision in mind. I have also seen first hand a large, “professional” company having large scale cybersecurity issues. For example, I had a job as a pharmacy technician and all of the centralized insurance databases across the nation were hacked by an unknown party, leading to mass shutdown of about a third of the systems we needed to do our job. This causes havoc and chaos for about 2 and a half weeks not just for our pharmacy, but for every retail, hospital, and specialty pharmacy in the US. Mistakes of this magnitude is not rare, which is why at every level of development we must keep in mind our ideals of precision and technicality.