**Summary**

Describe your unit testing approach for each of the three features.

**To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

My approach was to ensure that the high-level requirements were met first, then the more detailed oriented requirements were handled. I tested the functionality of the application in terms of adding, removing, and updating contact information. These were the most basic requirements of the application and I needed to ensure that they worked properly. Next, testing the details of the data was a little more difficult. More specifically, we were given character limits, updating preferences and whether or not a specific field was needed or could be null. These aspects of the application were difficult for me to understand, in terms of how to verify and enforce.

**Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

The Junit tests were complex and varied in level of difficulty for each program file. For the Contact test file, I was able to create the tests, but did not include the verification of the requirements for the program. I failed to consider the fact that I needed to declare the same requirements verification logic again, even though it was specified twice in the Contact file. The repetitiveness of the verification of requirements is still confusing to me. For the Task test file, a few of my tests had errors and one of my methods was not appropriately defined. For the Appointment test file, the tests also had errors and I was unable to resolve the logic issues.

**Describe your experience writing the JUnit tests**

**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate**

My ‘Contact.java’ was designed and developed to meet the specifications of the scenario in terms of classes, fields and data types. The ‘ContactService.java’ code was designed and developed to account for currently existing contacts, based upon the unique contactID field, and allowing for it to be created if it doesn’t currently exist. Additionally, similar types of logic were developed to allow for the deletion of contact information and updating contact information. For example, the following code allows for the deletion of a contact based upon the contactID.

This code searches for the unique contactID and if found, it is deleted and displays a message of its success, otherwise, it displays a message stating that it can’t be found:

/\* removes a contact based upon contactId if it exists in our contacts list \*/

public boolean remove(String contactID) {

for (Contact c : contacts) {

if (c.getContactID().equals(contactID)) {

contacts.remove(c);

System.out.println("Contact removed Successfully!");

return true;

}

}

System.out.println("Contact not present");

return false;

}

**How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate**

I ensured that my code was efficient by utilizing clear and concise coding practices that involve obvious syntax logic and methods. Through these functions, other developers can efficiently review the code, understand what’s happening and correct or update it to account for new requirements and necessary modifications. For example, when adding a new contact, we verify that it doesn’t already exist within our list by the contactID, and it does exist, it can’t be added, otherwise it will be. The code below conspicuously shows that logic:

/\* this method adds a contact to our list if it doesn't exist \*/

public boolean add(Contact contact) {

/\* if the contact is already present \*/

boolean alreadyPresent = false;

for (Contact c : contacts) {

if (c.equals(contact)) {

alreadyPresent = true;

}

}

/\* if the contact does not exist, we add it, and return true \*/

if (!alreadyPresent) {

contacts.add(contact);

System.out.println("Contact Added Successfully!");

return true;

} else {

System.out.println("Contact already present");

return false;

}

}

**Reflection**

Testing Techniques

**What were the software testing techniques that you employed in this project? Describe their characteristics using specific details**

Each milestone contained various tests based upon the scope of the program itself. In the first milestone, Contact, we tested the contact functionality, which consisted of verifying data types and lengths, as well as validating creating, deleting and updating the information for each contact. In the second milestone, Tasks, we performed similar tests as Contact (data types and field lengths), as well as creating, deleting and updating by ID. In the third milestone, Appointment, the same scope of tests we carried out. All of these tests were done so in a white-box testing fashion, meaning that they were based upon knowledge of the internal logic of the applications’ code (Garcia, 2017).

**What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details**

Other software testing techniques include black-box and non-functional testing. Black-box testing is based upon requirements without knowledge of the applications’ internal structure or code (Garcia, 2017). Non-functional testing is based upon system metrics which may include performance, load, volume and stress testing, all of which examines the programs ability to function appropriately as outside forces are employed to it.

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

For black-box testing, some techniques include systematic, random, GUI, model-based, smoke and sanity. For example, random testing involves testing the system across the entire input domain, and in some cases, mutating well-formed input data to examine the programs ability to reject malformed data. Additionally, sanity tests focus on ensuring that the basic features of the program work as expected before more complex components are tested.

For white-box testing, some techniques include code coverage, fault injection and mutating testing. For example, fault injection involves placing faults into the software infrastructure to examine how well or poorly the system behaves. Additionally, mutation testing involves examine the effects of data across several copies of the program, all of which contain small changes, which helps with examines code issues.

For non-functional testing, some techniques include performance, load, volume, stress, security, usability and accessibility. For example, volume testing focuses on examining the systems’ ability to continue processes and operations as the volume of data fluctuates. Additionally, usability examines the user experience, which may determine the interface to be difficult to use and understand.

Mindset

**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? Provide specific examples to illustrate your claims**

I employed caution throughout this project from the perspective of the end users. I wanted to ensure that my testing data and mindset were aligned with potential scenarios that the program might encounter, to appropriately account for various input data forms. For example, evaluating tests that accounted for ‘firstName’ values that exceeded ten characters, or ‘phoneNum’ values that were not exactly ten characters or ‘address’ values that exceeded 30 characters, helped us not only meet the requirements of the application but also anticipate potential use cases for data that would be rejected by the program logic.

It was important to appreciate the complexity and interdependencies between the program files because although the program was simple, the required attention to detail and consistency was crucial to the success of its operation, but also to my understanding of the concepts. Applying such concepts and principles in the real-world will be similar in expectation, but more valuable in the sense that it might be within the production environment, once deployed.

**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? Provide specific examples to illustrate your claims**

I can appreciate the need for external testing of code, as from the developer perspective, my eyes have scanned the code hundreds of times and as I became more familiar with its information, the less I became aware of changes and potential errors. Executing testing and QA assessments through fresh eyes will most certainly increase the odds of catching mistakes I’ve missed.

**Finally, 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? Provide specific examples to illustrate your claims**

It is imperative that developers take pride in their work during not only the development phases but also during the testing phases. Delivering functional code is obviously important to the client and their users, but delivering a program that is behaving in such a manner that it meets and continues to meet the expectations of its users solidifies our trust with the client and helps us build rapport and routine when it comes to discipline and ownership of our products. Testing accurately and often is one way to build a routine that avoids technical debt, one principle I intend on implementing from now on and hope to increase my productivity and skillset.

**References**

García, Boni. (2017). *Mastering software testing with JUnit 5*. Packt Publishing.