**Summary and Reflections Report**

Fathi Amran

SNHU

CS 302: Software Testing Automation

Kalysa Wilson

December 10, 2021

**Summary and Reflections Report**

In this reflection, I will describe the effectiveness of writing JUnit test cases, the mindset that helped me meet the client's requirements for the Contact, Task, and Appointment features, and the software testing techniques used. Throughout CS-320 Software Test Automation, I learned about the benefits of testing early and incrementally in the software development lifecycle. Furthermore, I was introduced to JUnit testing and the type of mindset software testing requires.

**JUnit Tests and Testing Mindset**

Before I began writing JUnit test cases, I needed to understand the client's requirements, which I did by carefully reading the client's instructions and asking for clarification in situations where the client's requirements were not clear or were conflicting. After understanding the client's requirements, I proceeded to write JUnit test cases based on my understanding of the requirements.

The client required three features, the Contact, Task, and Appointment services. Each feature had a different set of requirements; therefore, a different set of JUnit test cases were written for each feature. To help my readers understand the purpose of the test cases I developed, I will briefly list the client's requirements for each feature.

***Contact Requirements***

1. Has a unique ID that is not null, updatable, and larger than 10 characters
2. Has a first and last name that are not null, greater than 10-characters, and is updatable
3. Has a phone number that is not null, must be 10-digits long, and is updatable
4. Has an address that is not null, does not exceed 30-characters, and is updatable
5. Has the ability to add objects and update the updatable attributes or delete the object using the unique ID

***Task Requirements***

1. Has a unique ID that is not null, updatable, and larger than 10 characters
2. Has a name that is not null, longer than 20-characters, and is updatable
3. Has a description that is not null, longer than 50-characters, and is updatable
4. Has the ability to add objects and update the updatable attributes or delete the object using the unique ID

***Appointment Requirements***

1. Has a unique ID that is not null, updatable, and larger than 10 characters
2. Has a description that is not null, longer than 50-characters, and is updatable
3. Has an appointment date that is not null or in the past
4. Has the ability to add objects and update the updatable attributes or delete the object using the unique ID

**Meeting Client’s Requirements and Adopting a Mindset**

Now that I have discussed the client's requirements for the Contact, Task, and Appointment features, I will explain how my test cases have met the requirements and were technically sound, efficient, and effective. I will also describe my thought process and the mindset I adopted simultaneously with the JUnit test cases.

The biggest development challenge I experienced was to create a unique ID for each object that is not updatable, null, or exceeds the character limit. I had to think through my understanding of developing software to implement the application classes in a manner that makes testing easy and straightforward. I knew that there was a correlation between writing the application classes and writing test cases. The more complicated my code becomes, the more work and stress I create for myself in the testing phase.

My approach was to implement a vector of objects that simply stores every created object throughout the application. This vector needed to be static and have one instance in memory so that the created objects do not get stored in multiple vectors. The ideal implementation for the vector was to instantiate the vector when the program starts and to use it throughout the application. By taking this approach, I felt confident that all the objects I created were stored in the same place every single time. By this approach, I made searching for existing objects simple.

I heavily relied on the vector of objects to create unique IDs and update and delete the correct objects. Any object that got created was immediately added to the vector. When an update request was made, the vector was searched for the specified ID. If the object was found, the updating process proceeded; otherwise, the process was terminated, and an exception gets thrown notifying the user that the object ID does not exist. The same process took place when a request was made to delete an object. First, the vector was searched to find the specified ID. If the object was found, it gets deleted; otherwise, the process was terminated, and an exception gets thrown notifying the user that the object ID does not exist.

As an example to showcase the simplicity and ease of testing object deletion, the test case I implemented consisted of only one line: AppointmentService.deleteAppoitnment("12345").

All the logic of deleting an object resides in the delete methods of each feature. Below is the logic for the Appointment Service, but the approach is the same for the Contact and Task features. While examining the delete method, pay close attention to how the vector of objects was called statically and how it was then searched to find the specified ID. I also took my implementation a step further by removing all references to the deleted object so that the garbage collector can free the memory of the deleted object the next time it runs.

// delete appointment

public static void deleteAppoitnment(String ID) throws Exception {

// access vector of Tasks

Vector<Appointment> appointments = Appointment.appointmentIDs;

// find Appointment by ID

for(int i = 0; i < appointments.size(); ++i) {

if(appointments.get(i).getID() == ID) {

// delete object from vector so that

// the garbage collector can free memory

appointments.remove(i);

return;

}

}

// throw exception that ID was not found

throw new IllegalArgumentException("ID not found");

}

**Developer Bias**

Throughout development and testing, I eliminated my bias by adopting a skeptical mindset to remind me that my code is faulty by nature. By this approach, I kept my eyes open on faulty logic and complicated implementations that violate best coding and testing practices. When it was time to develop JUnit test cases, I played the role of a skeptical user, and when it was time to test the client's requirements, I played the role of an absent-minded user who does not abide by the software requirements. Playing the role of an absent-minded user allowed me to give the system wrong and unexpected input to see how it responds.

For example, if the software requirements asked for a 10-digit phone number, absent-minded me would provide a phone number with hyphens, and skeptical me would monitor the system's behavior. For instance, this check would pass the test:

assertTrue(Contact.instance().getPhoneNumber().equalsIgnoreCase("5597813348"))

But, this test would not:

assertTrue(Contact.instance().getPhoneNumber().equalsIgnoreCase("559-781-3348"))

In an ideal testing situation, a team of testers would hold responsibility for testing, but in this case, I had to play the role of the tester. I tried my best to eliminate bias by playing the role of a skeptical tester, and I eliminate technical debt by implementing the application in a way that follows the software industry's best practices. By following best practices, I developed the application with simplicity in mind to make testing easier.

**Software Testing Techniques**

I have already discussed the software testing techniques that I used and not used in a previous assignment. Instead of rewriting the same thing again, I will reference the mentioned assignment and provide snippets with minor edits.

I used the following software testing techniques: (1) functional testing, (2) regression testing, (3) integration testing, (4) system testing, (5) usability testing, and (6) acceptance testing. The software testing techniques that I did not use are (1) Black Box testing, (2) performance testing, (3) compliance testing, and (4) security testing.

Functional testing helped meet the client's requirements by testing the system’s object against the requirements. I tested the attribute's length and value, and the uniqueness of the IDs. Moreover, I tested the appointment date to ensure that only future dates were provided instead of dates in the past.

Regression testing helped me ensure that the code modifications I made to correct an error did not break other parts of the application. I heavily used this technique when making modifications to implement a vector of objects. I used integration testing to ensure that object creation, storing, and modification in either class were reflected throughout the whole program. I also used system testing to make sure the entire program functions properly as a whole.

I employed usability testing to make object creation, search, update, and delete straightforward. Lastly, I used White Box testing to test expected output, poorly written code, for-loop structure, and vector implementation. I used acceptance testing to ensure the program meets the client's requirements. I performed acceptance testing by playing the role of a user.

The following is excerpted from one of my previous assignments:

***Requirement Testing***

In other software applications, requirement testing encompasses different testing techniques to ensure the client's requirements and needs are met (Ganeshan, 2021).

***Regression Testing***

If an application had errors that got fixed, regression testing would ensure that the fix did not cause other errors to appear (Ganeshan, 2021).

***Integration Testing***

A typical system is normally comprised of multiple classes, modules, or sub-systems. When these components get integrated, they are tested to ensure compliance amongst each other.

***System Testing***

System testing in all applications ensures that the system as a whole, including its components and interface function as expected.

***Usability Testing***

A company would employ usability tests to measure the usefulness of its product.

***White Box Testing***

A company would use White Box testing to test the reliability of the system's inner workings, such as code structure, logic, and implementation.

***Acceptance Testing***

In acceptance testing, a company allows a user of the system or the client to test the software against the requirements. The purpose of this test is to give the client confidence that the produced application is satisfactory.

***Black Box Testing***

Black Box testing is a form of testing which checks the input and output of the system without having knowledge of its inner workings (Ganeshan, 2021). Because Black Box testing is normally performed by a third party, I did not perform this type of testing. In other software applications, a third party would test the system's input and output without having formal knowledge of the system's implementation.

***Performance Testing***

Performance testing falls under non-functional requirements and checks the system response, speed, and reliability under certain conditions (Ganeshan, 2021.). If a shopping application expects high system requests during promotional periods, performance testing would help measure the system's response to high-demand scenarios.

***Compliance Test***

Compliance testing ensures that software meets organizational, agency, and governmental standards. If an application is meant to serve users in California, then it must be tested to ensure that it gets the user's consent to sell their information.

***Security Testing***

Security testing, as the name suggests, tests the system's security defenses and vulnerabilities. A shopping application would incorporate security testing to expose the system vulnerabilities and to help protect the transfer of sensitive data.

**References**

Ganeshan, D. (2021, September 29). *Software testing techniques*. MST Solutions. Retrieved December 11, 2021, from https://www.mstsolutions.com/technical/software-testing-techniques/.