Brysson Schweitzer

CS-320 Software Test Automation & Quality Assurance

Southern New Hampshire University

10/09/2023

**Project Two – Summary**

Taking the software requirements of the project one assignment into consideration, my approach to this was to work on each of the requirements from the previous assignments and modify them to fit into the JUnit testing. For each of the assignments Contact, Appointment, and Task, I coded each of the requirements into the main class of each java package. Within each of the classes, I also wrote if statements to justify the requirements for each, including the lengths, cannot-be-null values, and to be updatable. This helped me align to the software requirements since I was meeting the requirements of each assignment by adding input validation for each and ensuring the variables were properly named.

When it came to beginning my JUnit tests, I had to rework them for project one to get each of the tests to pass above 80% as the requirement. To ensure my tests made it above 80% for each test, I ran multiple tests as I worked, so I knew where my code was getting stuck on in the test phase. Once I was able to work through the code and successfully get the test to pass, I would begin testing with the coverage as function. I knew my tests were effective based on the coverage percentage because I was able to reach 100% coverage for my test packages on all assignments, and above 80% for all my assignment packages.

While writing the JUnit tests, I initially struggled with the concept of the test function and had some errors in my @Test methods. During my first JUnit tests performed, I could only successfully test a portion of my code and reach close to 60% coverage. To fix this and ensure that my code was technically sound, I revisited and reviewed the previous module materials and located some resources that helped me to work on refining my code and JUnit tests to reach a much higher coverage. In a section of my JUnit test code:

**class** AppointmentTest { // defines the class AppointmentTest to test

@Test

**void** TestAppointment() { // tests the Appointment methods

Date appointmentDate = **new** Date();

Appointment appointment = **new** Appointment("000001", appointmentDate, "Description of test test test");

*assertTrue*(appointment.getAppointment\_ID().equals("000001"));

*assertTrue*(appointment.getAppointment\_Date().equals(appointmentDate));

*assertTrue*(appointment.getDescription().equals("Description of test test test"));

}

This section of code tests the appointment class method from the Appointment package and successfully provides results. The initial tests for my appointment class method did not return successful JUnit test results and failed to run the code in the @Test function. Rewriting my code in this test format helped me to ensure my code was technically sound and provided the proper results.

For me to ensure that my code was efficient, I had to structure my code in steps and rework my assignments to correctly define the argument exceptions that I needed to throw for answer validation. Included is a section of code that defined the illegal argument exceptions that I had to throw:

**public** Contact(String contact\_ID, String firstName, String lastName, String phone, String address) {

**if** (contact\_ID == **null** || contact\_ID.length() > 10) { // if loop that checks if the contact\_ID is null or greater than 10 characters

**throw** **new** IllegalArgumentException("Invalid Contact ID"); // throws an invalid contact ID exception

}

**if** (firstName == **null** || firstName.length() > 10) { // if loop that checks if the first name is null or greater than 10 characters

**throw** **new** IllegalArgumentException("Invalid First Name"); // throws an invalid first name exception

}

**if** (lastName == **null** || lastName.length() > 10) { // if loop that checks if the last name is null or greater than 10 characters

**throw** **new** IllegalArgumentException("Invalid Last Name"); // throws an invalid last name exception

}

This is only a part of the illegal argument exceptions that I needed to define for the Contact Java assignment. I wrote if statements to capture the requirements for the input validation. I also made sure I included comments in the code to help with efficiency for code review and to indicate what was happening in each section.

**Project Two – Reflection Report**

The software testing techniques that I employed for my project was the White-Box testing technique. This technique involves statement testing, which tests for a specific statement, and follows the definition to test if it is true or not. This also involves decision testing, which tries to force the program to execute decisions in controlled ways. The white-box test technique checks that each test section interacts correctly with the provided and test statements. This technique can also test whether a statement with an argument passes or fails the argument and relays the pass or fail status.

The other software testing techniques that I did not use for the project include the Black-Box testing techniques including Use-Case testing, State transition testing, decision table testing, or experience-based testing techniques such as exploratory testing and checklist-based testing. Decision table testing involves defining the specifications and conditions where each function operates in a table format. State transition testing is like decision table testing, but instead, we test the output triggers by changing the input conditions and it isn’t modelled in a table. Use-case testing involves capturing the interactions between the system and actors to the system and is typically used at the system level. Checklist-based testing involves a high-level checklist which is captured through a conjoining of individuals’ experiences and information from other sources. Exploratory testing is a technique that when used combines the experience of testers with the testing approach to maximize the amount of testing done within a time limit using test objects as its focus.

Decision table testing is used to test logical conditions and the effects of how each function operates. This is typically implemented to test business rules to define the functions of the system and to test the input conditions that produce various actions. State transition testing is used to test the different output triggers by changing the input conditions. This is used to test all the possible results and all the possible conditions. Use-case testing is a way to capture the process flows of a business process. This is often used to define changes to a system and the interactions between the users and the system that is being defined. Use-cases are typically used to define the expected behaviors related to changes involved with the system. Exploratory testing and checklist-based testing aren’t entirely reliable in that they derive from experience-based inputs and processes. Exploratory testing is normally used in testing more important areas of a system when other test methods can’t be used. Checklist-based testing is normally used with larger teams and isn’t widely used over other forms of testing.