**Summary and Reflections Report**

Ezra Maynard

University of Southern New Hampshire

Course Code: CS 320 Software Test Automation & QA

Ramsey Kraya, Ph.D.

Sunday, February 19, 2023

I tried my best to align the Junit tests with what were the requirements for my classes. For instance, in the Contact module, we were required to have our contact object have a String for firstName that could not exceed 10 characters. Because of this, I created a test firstName JUnit test that would test what happened to a firstName under 11 characters and one that threw an exception when the String exceeded 10 characters. I made sure to prompt the user to inform them they needed to update the field. I followed the class requirements in the assignment and made sure each requirement worked as it should both by having input validation in my class and testing various inputs using my input validation in the JUnit tests.

In an attempt to try to cover as many test cases as possible in my Task classes, I added input validation that would throw an error if the input did not fit into the same classification and length. I tested both a correct input and one that would throw an error to ensure that both test cases were handled correctly. To ensure the tests were effective, I made sure to employ both black box testing, which involves testing inputs versus their expected outputs. I also made use of white box testing by reading my code and making sure my logic was correct and functional.

if (id.length() > 10 || id == null) {  
 throw new IllegalArgumentException("Invalid input");  
}

Logically my code was as sound as I know how to make it. I didn’t use any overly complicated names or functions

private String id;  
private String taskName;  
private String taskDescription;

And my use of if-else statements allows for minimum issues with an input not fitting the correct String variable type and not being the correct length. In the below example, the value must be the correct length and cannot be empty or null, it cannot be anything other than a String. If anything of those things is not true, it throws an exception and lets the user know the input is invalid.

public Task(String id, String taskName, String taskDescription) {  
 if (id.length() > 10 || id == null) {  
 throw new IllegalArgumentException("Invalid input");  
 }  
  
 if (taskName == null || taskName.length() > 20) {  
 throw new IllegalArgumentException("Invalid input");  
 }  
  
 if (taskDescription.length() > 50 || taskDescription == null) {  
 throw new IllegalArgumentException("Invalid input");  
 }  
 this.id = id;  
 this.taskName = taskName;  
 this.taskDescription = taskDescription;  
}

To improve efficiency I tried to reduce the time complexity and memory usage by reducing the amount of if else loops and the overuse of the ArrayList. With the if-else statements I do have in my code, they are not lengthy and there’s typically only one loop, I made sure to use an if statement and a variable to track if the input passed certain parameters and would only change the variable to true if the input passes all checks. An example from my code would be the use of:

boolean updated = false;  
for (Task t : data) {  
 if (t.getId().equalsIgnoreCase(id)) {  
 t.setTaskName(taskName);  
 updated = true;  
 break;  
 }  
}  
return updated;

The variable “updated” only turns true if the Id passed into the function is found in our ArrayList.

One of the software testing techniques I have utilized is black box testing. It was very beneficial for me to check various inputs against what I thought the behavior of the code would be. I was able to find several issues in my code and it aided me in troubleshooting the Date instance situation I had. I thought I was inputting a correct Date format but Date doesn’t work like that and needs instances called for it, I was able to figure this out by seeing an exception thrown when I tried inputting a past date. I took the requirements of the program and classes and used the JUnit tests to verify that all the input validation was working properly.

I attempted to make use of equivalence partitioning and input partitions to try to validate my inputs as much as possible. For example, if an input needed to be below ten characters long, I made sure to break the input into three partitions. One less than ten characters, one at ten characters, and one above ten characters to ensure the largest amount of cases could have coverage through these tests.

One of the software techniques I did not utilize in any of my milestone assignments was state transition diagrams. I did not think they applied to my use case in the milestones but I could see them being incredibly insightful and useful on many real-world projects primarily websites, where you have state transitions based on user input in many areas. Having a visual representation of how the state transitions can be incredibly useful in creating efficient systems.

The practical uses for integrating input partitions into testing include cases where we have more options for incorrect inputs than valid inputs. For example, if we had an input that needed to be between 1-10. We only have ten correct inputs and millions of incorrect inputs, but input partitions make getting good coverage with minimal code very easy. In these cases, we must try to make sure we have tested our program against all known non-valid inputs. Input partitions allow us to do this by reducing the number of test cases we have to create by partitioning the input into chunks that are grouped into groups of similar input. Without this, we would have to write thousands of test cases for input validation checking.

When it comes to mindset, the one I adopted while working on this project was caution and awareness of how complex the code could get. The more error checking and input validation within functions we have the more usable to code will be. I tried my best to limit bias while reviewing my code and trying to see it as someone else's code and not think of it as my own. As developers, we must try to stray from bias when it comes to evaluating our code. It’s always helpful to have a second pair of eyes or even more review the code with you to ensure there is minimal bias.

It’s important to stay disciplined as a developer because we need to be committed to making secure and efficient code that has been tested properly. If we’re not careful our code can cause some pretty serious consequences that are typically time and money, however, they can also be lethal or life-threatening as well. I will always create JUnit tests for Java, review my code with minimal bias, and I will try my best to set an example for others in the field on how testing should be done.