**CS-320 Project Two**

**Petar Petreski**

**1. Summary**

My testing approach was very much aligned with the software requirements. All requirements were closely analyzed and considered, and my testing approach was tailored to ensure that all requirements are tested and covered. For example, the task class requirements were that the task ID String cannot be longer than 10 characters, shall not be null and shall not be updatable. So, I tested the task object for these requirements and told it to throw IllegalArgumentException if the requirements are not met:

public Task(String ID, String taskName, String taskDescription) {

if(ID == null || ID.length()>10) {

throw new IllegalArgumentException("Invalid ID");

}

I have done the same tests for other objects (task name, task description, contact first name, contact last name, and so on).

The overall quality of my Junit test for the task service is perfect at 100% which means that all requirements were met, because we were testing the task service class for the requirements. The coverage for the contact service was pretty low which means that I didn’t do a good job at testing the contact service requirements, but since then I re-did the tests and current coverage for the contact service class is also at 100% which means that all requirements were met and Junit tests were effective. I will include the improved service class test in Project One.

To ensure that my code is technically sound I created a Task class and defined all its methods (getters and setters) and tested them for requirements. All this made this code clean, readable and maintainable. For example:

public class Task {

private String ID;

private String taskName;

private String taskDescription;

public Task(String ID, String taskName, String taskDescription) {

if(ID == null || ID.length()>10) {

throw new IllegalArgumentException("Invalid ID");

public String gettaskName() {

return taskName;

}

public void settaskName(String taskName) {

if(taskName == null || taskName.length()>20) {

throw new IllegalArgumentException("Invalid task name");

}

this.taskName = taskName;

}

To ensure that my code is efficient I tried to write code with as fewer lines as possible resulting in a code that is shorter, clearer and maintainable. Choosing the right data structure is the first important step in ensuring code efficiency. We chose a map as a better choice instead of a list because it offers a collection of key/value pairs:

public class TaskService {

private Map<String, Task> mapOfTasks = new HashMap<String, Task>();

// adding a new record

public void addTask(Task task) {

if (mapOfTasks.containsKey(task.getID())) {

throw new IllegalArgumentException("Task ID must be unique");

}

mapOfTasks.put(task.getID(), task);

}

**2. Reflection**

We started our testing by identifying test conditions and specifying test cases. For example, the contact object, task object, or appointment object cannot be longer than 10 characters and shall not be null. This means that we used a so-called black-box technique, also known as functional testing, because it’s based on the specific functionality of the object to be tested. We don’t know the internal program structure with black-box technique, but its behavior can be determined by observing its inputs and related outputs. To test the internal program structure, we used white-box or structural testing. Some of the white-box techniques we used were code coverage, fault injection, and mutation testing. We used statement coverage to measure the percentage of statements executed. So, in order to select a good set of test cases, we used a conjunction of both black and white-box techniques.

One of the testing techniques I did not use for the project is non-functional testing. Non-functional testing is used to test if the system under test meets specific performance, load, volume, or stress requirements. Also, I didn’t use experience-based techniques which are based on the testers' experience and not on specification-based test cases. Understandably, I didn’t use this technique, considering that I don’t have any previous experience in testing.

The black-box testing technique is very practical because it doesn’t require the testers to see or even understand the code they’re testing. Black-box keeps the code hidden which also preserves intellectual property. White-box testing shows how the internal working of the code was performed. This is helpful because it doesn’t just test if something works or not, but also tests for all called classes and methods. Non-functional testing is also practical and important because it improves software’s performance in terms of speed and security which ultimately improves user experience.

While working on this project, I focused on being cautious and creating the right testing strategy. As a tester, I had to ensure that all classes and objects were tested thoroughly to meet the requirements or uncover potential issues, and that is why it was important to appreciate the complexity and interrelationships of the code I was testing.

In this project, we were required to write the code that would meet the requirements, but at the same time, we were required to test the same code to ensure that it meets the requirements. This is why we had to check our code objectively and limit the bias in our code review. The developers may become too attached to their code, or too confident in their code, which may affect their bias. Even though I was confident that my written code met the requirements, I had to put the “tester hat” on and approach the testing phase as I had never seen the code before. There is a difference in the mindset; the software developers try to make the code work while the testers try to break it, and that is why bias is a concern if you are responsible for testing your own code.

All this brings us to the importance of being disciplined as a software engineering professional. Being disciplined is very important in all spheres of life in general, but in this technological age and future, the consequences of not being disciplined as a software engineering professional can be catastrophic in terms of resources or even human lives. There are many examples in the past where the developers’ or testers’ mistakes were a direct cause of a disaster, that being the development of the F35 fighter jet, the Ariane 5 rocket explosion, or the loss of Chinook ZD576 with its 25 passengers. This is why we need to ensure that we as professionals possess discipline, moral compass, and integrity that will allow us to provide an adequately tested code.

References:

Boni Garcia. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.

*Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. (2019). BCS Learning & Development Limited.