**Project Two**

Astrid French

Department of Computer Science, Southern New Hampshire University

CS-320-R4840: Software Test Automation & QA

Omar Toledo Lopez

17 April 2024

Summary and Reflection

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

Understanding the software requirements is crucial for guiding the coding process and ensuring that the software fully meets the specified requirements. Each requirement should be broken down into detailed specifications to achieve bug-free software.

For example 1, consider the "ContactService" project. One of the requirements states:

"The contact object shall have a mandatory 'phone' field that must consist of exactly 10 digits. The 'phone' field cannot be null. It is required that the input provided to the software is in numeric format."

This requirement specifies that the 'phone' field of the contact object must contain exactly 10 digits and cannot be empty. In this code, there is a condition where the phoneNumber is null or not 10 digits. It will throw new exception and print out “Invalid Phone Number”. Additionally, it emphasizes that the input provided to the software must be numeric to fulfill this requirement. This is what we called black box testing where we make sure the software accept the correct input.

//phoneNumber have to be a digit and 10 digits  
if(phoneNumber == null || phoneNumber.length() != 10 || !isValidPhoneNumber(phoneNumber)){  
 throw new IllegalArgumentException("Invalid phone Number.");  
}  
this.phoneNumber = phoneNumber;

// Method to check the phone number only of numeric digits  
private boolean isValidPhoneNumber(String str){  
 //if the phone number is null and the digits is not 10  
 if (str == null || str.length() != 10) {  
 //return false  
 return false;  
 }  
 //if the phone number is not null and the digit is 10  
 for (char c : str.toCharArray()) {  
 //to check if the character is not numeric(number)  
 if (!Character.*isDigit*(c)) {  
 return false;  
 }  
 }  
 return true;

Example 2:

For the TaskService Project, the requirement is as follows:

"The task service must be capable of updating task fields based on the task ID. The following fields are updateable:

Name

Description

In this requirement, we understand that only the task ID can be used to update these fields. We need to ensure that our code adheres to this requirement by allowing input of the ID to update the specified fields. Our code will generate and search for the ID, and once a match is found, we can update either the task name or the description. This is what we called black box testing where we make sure the software accepts the correct input.

//method to update the task name by task ID  
public boolean updateTaskNameField(String taskID, String newNameField){  
 //generate the list of task ID  
 for(Task task : list){  
 //if the task ID is matched  
 if(task.getTaskID().equalsIgnoreCase(taskID)){  
 //set the new name  
 task.setNameField(newNameField);  
 return true;  
 }  
 }  
 return false;  
}  
  
//method to update the task description by task ID  
public boolean updateTaskDescField(String taskID, String newDescField) {  
 //generate the list of the task ID  
 for (Task task : list) {  
 // if the task ID is matched  
 if (task.getTaskID().equalsIgnoreCase(taskID)) {  
 //set the new description  
 task.setDescField(newDescField);  
 return true;  
 }  
 }  
 return false;  
}

Example 3:

For the Appointment Service, one of the requirements is as follows:

"The appointment object must include a mandatory appointment date field. This appointment date field cannot be set to a past date and cannot be null.”

From this requirement, we understand that only future dates and times should be used, inputting past dates should be prevented to avoid bugs. We want to ensure that customers cannot schedule appointments for dates that have already passed.

To achieve this, we need to incorporate time validation into our code.

Using import java.util.Date;

And write condition as:

if (appointmentDate == null || appointmentDate.before(new Date())) {  
 throw new IllegalArgumentException("Invalid Date");  
}  
this.appointmentDate = appointmentDate;

In this scenario, if the appointmentDate is null or if the appointment date occurs before the current date or time, an exception will be thrown with the message "Invalid Date". Otherwise, if these conditions are not met, the appointmentDate will be assigned.

* + 1. 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?

I am using Junit testing techniques in the ContactService Project, TaskService Project, and AppointmentService. Each project has specific requirements and Junit test covered the software 100%. For example, in the Appointment class, it is crucial to ensure that IDs cannot be null and must not exceed 10 characters. Additionally, the appointment date must be in the future, and the appointment description must not exceed 50 characters and cannot be null. Every entity introduced in my code undergoes thorough testing to ensure compliance with these requirements. On the AppointmentService class, tasks for each software, such as adding an appointment and deleting the appointment by ID, are rigorously tested against their respective requirements. If any requirement is not met, I refactor the code accordingly.

I thoroughly tested each requirement. Additionally, I adjusted constructors where specific members have particular requirements, such as maximum character limits, disallowance of null values, and in the Appointment Service project, I imported the Java standard library, which includes functions to handle dates. I also ensured that the date cannot be set in the past. Furthermore, I declared the ID as a private final string, preventing it from being updated.

I use methods provided by JUnit such as assertFalse, assertTrue, and assertThrows. assertFalse verifies that a condition is false, and the test passes if the condition is indeed false. If the condition is not false, the test fails. Similarly, assertTrue verifies that a condition is true, and the test passes if the condition is true. If the condition is not true, the test fails. On the other hand, assertThrows is used to verify that a specific exception is thrown under certain conditions. If the condition is met and the expected exception is thrown, the test passes. However, if the condition is not met or the expected exception is not thrown, the test fails. This approach helps ensure that the code behaves as expected under different conditions and exceptions are handled appropriately.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

I ensured that my code was technically sound by conducting thorough JUnit testing and ensuring that it met all the specified requirements. For each requirement, I implemented corresponding test cases to validate its functionality.

**First testing example:**

@Test  
public void testAppointmentIDisTooLongFails(){  
 //Create a date for appointment July 3, 2024  
 Date appointmentDate = new Date(124, 6, 3);  
 Assertions.*assertThrows*(IllegalArgumentException.class,()->{  
 new Appointment("AstridFrench",appointmentDate,"Description");  
 });  
}  
  
@Test  
public void testAppointmentIDisNullFails(){  
 //Create a date for appointment July 3, 2024  
 Date appointmentDate = new Date(124, 6, 3);  
 Assertions.*assertThrows*(IllegalArgumentException.class,()->{  
 new Appointment(null, appointmentDate,"Description");  
 });  
}

On Appointment Project.

* The appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.

Checking the requirements themselves to verify if they work or not. For example, in this code, if the appointment ID is too long, it will fail. We also check what happens if the "ID" is null.

**Second testing example:**

On Task Project.

* The task service shall be able to update task fields per task ID. The following fields are updatable:
  + Name
  + Description

@Test  
 void testUpdateTaskNameWithTaskIdSuccess(){  
 //Create new instance  
 TaskService taskService = new TaskService();  
 Task task1 = new Task("AstridF","The Hamster",  
 "The Hamster lifepans is only 2-3 years");  
 //Add the task  
 *assertTrue*(taskService.addTask(task1));  
 //Updated the newNameField by the taskID  
 *assertTrue*(taskService.updateTaskNameField("AstridF","The Dog"));  
 }  
  
 @Test  
 void testUpdateTaskDescWithTaskIdSuccess(){  
 //Create new instance  
 TaskService taskService = new TaskService();  
 Task task1 = new Task("AstridF","The Hamster",  
 "The Hamster lifespan is only 2 - 3 years");  
 //Add the task  
 *assertTrue*(taskService.addTask(task1));  
 //Updated the newDescField by the taskID  
 *assertTrue*(taskService.updateTaskDescField("AstridF",  
 "Hamster is the best pet I ever have."));  
 }  
}

In this Junit testing, I tested if we update the task name by task ID and task description by task ID. If they are success its going to assertTrue and continue with the new name or description.

**Third testing example:**

@Test  
void testGetAppointmentDeletedWithIdSuccess() {  
 // Create a date for appointment July 3, 2024  
 Date appointmentDate = new Date(124, 6, 3);  
 AppointmentService appointmentService = new AppointmentService();  
 Appointment appointment1 = new Appointment("AstridF", appointmentDate, "Description");  
 *assertTrue*(appointmentService.addAppointment(appointment1));  
 *assertTrue*(appointmentService.deleteAppointment(appointment1.getAppointmentID())); //Deleted by the appointment ID  
}

Checking the software works as requirements. In this example, I tested to get my appointment deleted by ID and it succeed.

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

 My code was efficient because:

* I prioritized simplicity and minimized complexity in my code, ensuring it remains concise and straightforward. Each declaration is accompanied by comments explaining its purpose, facilitating understanding for other readers. I avoided code repetition to enhance readability and maintainability.
* I focused on including only essential and necessary variables, ensuring they adhere to the project requirements. Additionally, I diligently verified the correctness of each variable declaration.
* I imported Java libraries such as import java.util.Date; to assist in handling date-related operations, which helped streamline the implementation and improve the efficiency of my code.
* I utilized a linear search algorithm in certain parts of my code where I needed to find an ID that matches a previously saved ID.
* // Iterate through each taskID in the list  
  for (int i = 0; i < list.size(); i++) {  
   // Get the task ID at the current index  
   Task currentTask = list.get(i);  
   // Check if the ID of the current contact matches the specified taskID  
   if (currentTask.getTaskID().equalsIgnoreCase(taskID)) {}
* I conducted thorough testing of my code at various stages, checking each method's functionality to prevent potential errors in the future.
* I adhered to the camelCase naming convention for variables, ensuring that their names accurately reflect their functions. For instance, "appointmentDesc" is used to denote the description of an appointment.

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

**Software testing technique that I employed in this project:**

* JUnit testing: I conducted individual tests for each requirement to ensure that each one was fulfilled. I also explored various input scenarios to uncover potential bugs. For example, in the ContactClass Project, while testing the phoneNumber field, I initially assumed that only 10 digits would be accepted. However, through testing, I discovered that the field also accepted characters such as alphabets. This insight significantly impacted the testing approach and helped identify potential issues that would have otherwise gone unnoticed.
* Regression testing: I performed regression testing multiple times after making any changes or additions to the code. This ensured that the code continued to run smoothly and that no new bugs were introduced inadvertently.
* Black box testing: I focused on ensuring that the software accepted only the correct input. It's crucial to prevent users from entering random or incorrect input that does not align with the software's requirements. For example, in the case of the phoneNumber field, I enforced the rule that only up to 10 digits were accepted, as anything beyond that would be invalid.
* Equivalence partitioning testing: I employed equivalence partitioning testing to group input into categories, thereby reducing the number of test cases required. This approach allowed for efficient testing coverage while minimizing redundancy in the test suite.
* Decision testing was frequently employed within my loops to verify the ongoing compliance of input conditions. If the conditions were not met, the input would be discarded.
  + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

Here are some software testing techniques that I did not use for this project:

* White box testing: This testing method examines the internal structure of the code itself.
* System testing: This involves testing the software or application as a whole to evaluate its overall performance.
* Integration testing: This technique tests the interactions between different components to ensure they work together seamlessly. Since there are three separate projects in this case, I did not have the opportunity to integrate them together for testing purpose.
  + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.
* **JUnit testing: It helps to test each individual requirement. In my project, I used JUnit testing extensively to test each requirement.**
* **Regression testing: It helps to track changes in code results over time. We need to rerun the code every time there is a change to ensure no significant errors occur in the future. “Using appropriate regression testing tools eliminates software defects early in the software deployment cycle” (Kanade, 2022).**
* **Black box testing: It ensures that all inputs to the software are accepted and function as expected. It examines all inputs into our software.**
* **Equivalence partitioning testing: It saves time and effort by reducing testing within groups.**
* **Decision testing: It ensures the software only accepts certain conditions, and if not accepted, it moves to another step. If not accepted, it will be discarded.**
* **White box testing: It tests the internal structure of the code itself, preventing any coding errors.**
* **System testing: It ensures that the entire software works and meets user requirements.**
* **Integration testing: It ensures that the entire software, when combined, works seamlessly and is bug-free.**
  1. Mindset
     1. 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.

**The mindset I adopted while working on this project focused on ensuring that the software was free of bugs and aligned with the specified requirements. The testing techniques I employed, such as JUnit testing, regression testing, black box testing, equivalence partitioning testing, and decision testing, were instrumental in achieving this goal. Additionally, paying attention to coding practices, such as avoiding redundancy, enhancing the quality and reliability of the codebase.**

**It's important to appreciate the complexity and interrelationships within the code I was testing. As I mentioned, the testing techniques I employed were complex and interconnected.**

**For example, without black box testing to validate that the correct input is fed into the software, JUnit testing wouldn't be able to accurately test the functionality. Consider the phoneNumber field in the ContactClass Project. It's essential to ensure that only numbers are accepted as input, and black box testing verifies this requirement. Then, JUnit testing can be used to test the input against various scenarios.**

**Similarly, regression testing was vital. We needed to rerun our code after every change to minimize the risk of introducing errors or regressions. This continuous testing approach helps maintain the integrity and stability of the software over time.**

* + 1. 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.

Never assume that your code is flawless, always prioritize testing. The more testing techniques you employ, the better equipped you are to prevent potential bugs in the future. It's advisable to allocate time for testing, as thorough testing can uncover hidden issues and enhance the overall quality of the software. “A properly tested software product ensures dependability, security, and high performance, which leads to time savings, cost effectiveness, and customer satisfaction” (Parthiban, 2021).

It's important to begin testing early in the development process, even when the codebase is relatively short. Testing early allows you to identify and address issues promptly, preventing them from escalating into more significant problems later on.

* + 1. 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.

**Being disciplined in commitment to quality as a software engineering professional is important for several reasons. First, it's essential to uphold the integrity of the software and ensure it meets the specified requirements while being free of bugs. However, beyond meeting functional requirements, maintaining quality is important for earning the trust of customers and users. Discipline is reflected in the quality of your code. Follow coding standards, write clean and maintainable code, and adhere to best practices. “Document your code thoroughly to aid in collaboration and future maintenance. Code reviews and automated testing should be an integral part of your development process” (Abiodun, 2023).**

**For example, consider a scenario where a software engineer fails to thoroughly test a banking software update (cutting corner). Due to this oversight, a critical bug goes unnoticed, causing the interest rate calculation to malfunction. Instead of applying the intended 1% interest, the software incorrectly applies a 5% interest rate. As a result, leading to a loss of trust in the bank's services and tarnishing its reputation.**

**References:**

Abiodun, I. (2023). How to become disciplined as software engineer?. Medium.

https://medium.com/@engrisaac/how-to-become-disciplined-as-software-engineer-10944c55126c

Kanade, V. (2022). *What is regression testing? Definition, techniques, and tools*. SpiceWorks.

https://www.spiceworks.com/tech/devops/articles/what-is-regression-testing/

Parthiban, P. (2021). *7 Reasons why software testing is important*. Indium.

https://www.indiumsoftware.com/blog/why-software-testing/