**Armando Gomez**

**CS-320-R3321 Software Test Automation& QA 24EW3**

**7-2 Project Two Summary and Reflections Report**

**Professor Jonathan Norman**

**February 23, 2024**

**Summary**

**Describe your unit testing approach for each of the three features.**

My method for conducting unit testing was carefully planned to guarantee the reliability and performance of the three features: appointments, contacts, and tasks. Each unit required a testing approach to address its distinct needs and potential exceptional scenarios.

For the Appointments Unit test, my focus was on validating operations like creating, updating, and deleting appointments. The tests were specifically designed to ensure that only appointments met the criteria. Such as having valid identifiers, future dates, and descriptions within predefined parameters. I conducted a series of tests that examined date validations, including scenarios like leap years and time zone differences, to confirm the system's capability in handling date-related logic.

For the Contacts unit test, I customized the testing process to thoroughly validate adding, modifying, and removing contact information. Considering the sensitivity and significance of the data, I included tests to ensure each contact could be uniquely identified and that all personal details adhered to specific formatting and length standards. This included input validation tests to guarantee that the system promptly rejected any invalid data to uphold the accuracy of contact information.

For the task unit test, a targeted approach was necessary to assess functionality related to task management effectively. I made sure that tasks could only be created with IDs and had meaningful names and descriptions. I paid attention to checking inputs to reject null values or entries that were too long. Additionally, I thoroughly tested the update and delete functions to make sure they worked correctly and maintained the integrity of the task lists.

Throughout the testing phase, I took a thorough approach, ensuring that each test case matched the specific requirements of the feature it was testing. This thorough method of unit testing for all three features played a role in building confidence in the application's reliability and performance.

**To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

During the project, one key focus was to make sure that only appointments for dates could be scheduled. To achieve this, I created a set of tests like "testFutureDateValidation()" to try making appointments with future dates. These tests aimed to confirm that appointments in the past were rejected while those in the future were accepted. The successful completion of these tests, where only future dates passed validation, provided proof that our testing method aligned with the software requirements.

For the Contacts test, it was essential that all contact phone numbers met a format requirement. To address this, I designed tests such as "testPhoneNumberFormatValidation()" to input phone number formats into the system. The goal was to ensure that only phone numbers matching the required format were approved. The consistent rejection of formatted phone numbers by these tests clearly demonstrated our testing strategy's compliance with the specified requirements.

The Tasks unit test, a crucial requirement was keeping task descriptions brief. Within a set character limit. To verify this, I carried out tests like "testTaskDescriptionLengthValidation()" to add tasks with description lengths. During these tests, it became evident that the system effectively adhered to the character limit, highlighting how my testing method aligns well with the software requirements.

The test coverage metrics showed support for this alignment. The high percentage of coverage indicated testing of most parts of the codebase, including edge cases. This extensive coverage guarantees that the software functions correctly under conditions reinforcing the harmony between the testing approach and software requirements.

**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?**

The quality of my JUnit tests can be well supported by looking at the coverage percentage, which's a key measure of how effective the tests are. In this project, our JUnit tests had a high coverage percentage, showing that they were thorough and comprehensive.

A high coverage percentage means that a large part of the codebase was tested during testing. This is important because it shows that not only were the main functions of the application thoroughly tested, but also the less common scenarios and paths were considered. For example, having 97% instruction coverage in the services package indicates that all logical paths in the service layer were tested. This level of coverage is crucial in a layered application like this one, where business logic in the service layer can have a significant impact on how the application behaves.

A black screen with white text

Description automatically generated

Having 75% branch coverage in the package shows that most conditional code blocks were evaluated for both true and false outcomes, which is vital for testing decision-making logic in the application. While 100% branch coverage would be ideal, achieving 75% in applications with many conditional paths still demonstrates a strong and effective testing approach. The JUnit tests prove their effectiveness through tests created to validate essential requirements.

Tests ensuring that appointments can only be scheduled in the future directly address a business rule. The consistent success of these tests in upholding this rule across scenarios highlights their effectiveness in confirming the application's behavior aligns with specified requirements.

Not only were the JUnit tests designed to pass in optimal conditions, but they were also intended to handle failures gracefully under incorrect conditions, testing the application's resilience and error handling capabilities. The approach I took guarantees that the application functions securely even in unforeseen circumstances, further showcasing the quality of the tests.

With a high coverage rate and a strategic test design targeting critical functionalities and edge cases, the overall quality of the tests is strongly supported. These tests effectively ensured that the application fulfilled its requirements and behaved as intended across scenarios, showcasing their efficiency and thoroughness in the testing strategies I implemented.

**Describe your experience writing the JUnit tests.**

Writing JUnit tests for the project was a journey that enhanced my analytical skills and deepened my grasp of how the application works. Planning meticulously to cover scenarios, including those tricky edge cases, was quite a task. It was challenging to strike a balance between thoroughness and efficiency, needing consideration to avoid repetition while ensuring broad coverage. Making use of JUnit features like @BeforeEach and @parameterizedtest helped streamline the tests and keep them easy to maintain.

Troubleshooting failing tests played a role in this learning process, offering valuable insights into resolving subtle bugs. This exercise not only boosted my coding discipline but also underscored the significance of writing modular code and having robust error handling practices. Writing JUnit tests was demanding yet fulfilling, greatly shaping my growth as a software engineer by reinforcing practices and attention to detail.

**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

I used JUnit tests to check the effectiveness and reliability of the code. Like when I was testing the Appointments function, I created a test to verify that appointments couldn't be scheduled for dates.

@Test

void testAppointmentWithPastDate() {

Calendar pastDate = Calendar.getInstance();

pastDate.set(Calendar.YEAR, 2020); // Setting a past year

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Appointment("1", pastDate.getTime(), "Past Appointment");

});

assertTrue(exception.getMessage().contains("Invalid date"));

}

This test tries to schedule an appointment with a date that has already passed and anticipates an "IllegalArgumentException" to be raised with a message indicating a date. By using "assertThrows" and "assertTrue" to verify the exception message, it ensures that the code accurately validates appointment dates.

I made sure to adhere to coding practices by using clear variable names, keeping my methods concise and purposeful, and avoiding repeating code. For example, when working on the Contacts function, I utilized labels and organized the validation rules neatly.

public void setPhoneNumber(String phoneNumber) {

if (!validatePhoneNumber(phoneNumber)) {

throw new IllegalArgumentException("Invalid phone number");

}

this.phoneNumber = phoneNumber;

}

private boolean validatePhoneNumber(String phoneNumber) {

return phoneNumber != null && phoneNumber.matches("\\d{10}");

}

Using 'setPhoneNumber' makes its purpose evident, while 'validatePhoneNumber' contains the logic for verifying the phone number format, improving code readability and ease of maintenance.

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

In order to make the code more efficient in my project, I concentrated on improving performance and managing resources in the "AppointmentService" class. Here's my strategy for enhancing efficiency, along with references to code snippets from the "AppointmentService.java" file.

public static HashMap<String, Appointment> appointments = new HashMap<>();

I selected data structures to handle appointments efficiently. For example, I opted for a "HashMap" to store appointments, enabling retrieval, addition, and removal based on appointment IDs.

I optimized the logic to prevent extra processing. For instance, when inputting a new appointment, I made sure that the appointment ID was created in a manner that prevented overlaps, thus eliminating the need for extra verifications.

public void addUniqueAppointment(Date date, String description) {

String uniqueID = generateUniqueID();

Appointment newAppointment = new Appointment(uniqueID, date, description);

appointments.put(uniqueID, newAppointment);

}

When I created the “generateUniqueID()” function, it helped to guarantee that each ID is unique. This makes the process of adding IDs more efficient because it prevents any issues that may arise from ID conflicts.

**Reflection**

**Testing Techniques**

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

I focused on essential software testing methods to ensure a thorough examination of the application. Firstly, I relied on unit testing, which involves assessing code components independently. Within the "AppointmentTest" class, I implemented a test method called "testFutureDateValidation()" to verify the correct restriction of appointment dates to future dates, only preventing any past dates from being accepted.

I continued with integration testing to assess how these individual components interact with each other. An illustrative scenario would be examining how the "AppointmentService" class collaborates with Appointment objects to confirm that adding an appointment through the service accurately updates the system.

Also, I frequently utilized boundary value analysis (BVA) in cases related to input validation. In the Contact class, I conducted tests such as "testPhoneNumberLength()" to validate the phone number's length at its extremities and ensure handling of excessively long or short numbers.

Equivalence partitioning was instrumental in streamlining the testing process by categorizing inputs together.In the task file, tests such as "testValidTaskName()" were created to verify valid input names. The assumption was that if one name from this set succeeded, the rest would as well.

**What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.**

While I concentrated on unit and integration testing for Java files such as Appointment.java, Contact.java, and Task.java, I didn't utilize other testing methods because of the specific focus and requirements of the project. For example, performance testing, which could have been used to measure how efficiently "AppointmentService" manages appointments, wasn't implemented since the project emphasized functionality over load handling. Security testing, important for detecting vulnerabilities, was not prioritized, and the security of "ContactService" against threats was not explored. Usability testing, which assesses user interface design, was skipped as the project's main focus was on logic rather than user interfaces. Compatibility testing to test software performance across platforms was deemed unnecessary as the application was designed for a specific environment.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

I used a variety of software testing methods, like integration testing, boundary value analysis, and equivalence partitioning. These techniques are widely applicable in software development scenarios. Unit testing involves testing parts independently. For example, checking the "setDate()" function in the "Appointment.java" file to confirm that it accepts only future dates. This type of testing is essential for ensuring the dependability of core code components in projects ranging from application to intricate systems.

// In Appointment.java

public void setDate(Date date) {

if(date.before(new Date())) {

throw new IllegalArgumentException("Date must be in the future.");

}

this.date = date;

}

**Mindset**

**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.**

I took a careful approach by creating comprehensive test cases that spanned various scenarios, including more complex edge cases. When testing the "TaskService" class, I didn't just focus on scenarios like adding a new task with a valid name and description. My tests also explored conditions, such as adding tasks with names or descriptions at their maximum allowed length, to confirm that the system could handle such situations effectively.

// Example from TaskServiceTest.java

@Test

void testAddTaskWithMaxLengthDescription() {

String maxDescription = String.join("", Collections.nCopies(MAX\_DESC\_LENGTH, "a")); // MAX\_DESC\_LENGTH is a constant representing the maximum allowed description length

assertDoesNotThrow(() -> taskService.addUniqueTask("TaskName", maxDescription));

}

The effectiveness of the "TaskService" class heavily depended on how the functions in the "Task" class, like "setDescription()," performed. If there were any mistakes in the validation process of the "task" class, it could cause problems in managing tasks at the service level. That's why I made sure that the unit tests for the "Task" class thoroughly examined every validation guideline.

// Example from TaskTest.java

@Test

void testTaskWithInvalidDescription() {

String invalidDescription = String.join("", Collections.nCopies(MAX\_DESC\_LENGTH + 1, "a")); // Exceeding the maximum allowed description length

IllegalArgumentException thrown = assertThrows(IllegalArgumentException.class, () ->

new Task("1", "Urgent Task", invalidDescription));

assertTrue(thrown.getMessage().contains("Invalid description"));

}

**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.**

To address any bias in evaluating my code, especially when acting as both the developer and tester, I utilized various methods to uphold fairness and completeness. I heavily relied on automated testing, specifically utilizing JUnit tests to establish a standard for assessing the accuracy of the code. Within the "AppointmentServiceTest.java" file, I created tests to automatically validate the addition and removal of appointments, guaranteeing that my personal opinions did not impact the evaluation of functionality.

// In AppointmentServiceTest.java

@Test

public void testAddAppointment() {

AppointmentService service = new AppointmentService();

service.addAppointment(new Date(), "Checkup");

assertEquals(1, service.getAppointments().size());

}

@Test

public void testDeleteAppointment() {

AppointmentService service = new AppointmentService();

String id = service.addAppointment(new Date(), "Checkup");

service.deleteAppointment(id);

assertTrue(service.getAppointments().isEmpty());

}

It was important to have refactoring sessions. I would go back to parts of the code, such as the validation logic in Task.java, and carefully assess the efficiency and clarity of how it was implemented. This approach was effective in overcoming any bias towards familiarity and promoting enhancements.

// In Task.java

public void setDescription(String description) {

if(description.length() > MAX\_LENGTH) {

throw new IllegalArgumentException("Description too long.");

}

this.description = description;

}

**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.**

Maintaining a commitment to quality is crucial in the field of software engineering, as it directly influences the dependability, manageability, and overall success of software products. While taking shortcuts when writing or testing code may provide short-term advantages in terms of speed or workload reduction, it inevitably leads to debt. A collection of issues that will necessitate significantly more time and resources for resolution in the future. For example, rushing through testing for the "AppointmentService" class may accelerate its deployment process. Without thorough validation, it could result in unnoticed bugs related to appointment scheduling or deletion. This could lead to system failures or data inconsistencies that undermine user confidence and require corrections.

To prevent debt from accumulating, I intend to follow industry best practices like test-driven development (TDD), which involves creating tests before writing actual code. This not only clarifies the requirements but also ensures that the code aligns with these requirements right from the start. By creating tests for the "validateContact" method in the "ContactService" class before implementing it, I can confirm that each contact undergoes validation against specified criteria, reducing the chances of introducing errors during development.

**References:**

Gordon, T., & Makovhololo, P. (2023). The Application of Structuration Theory in Software Testing within the Organisation. *2023 International Conference on Electrical, Computer and Energy Technologies (ICECET), Electrical, Computer and Energy Technologies (ICECET), 2023 International Conference On*, 1–9. <https://doi-org.ezproxy.snhu.edu/10.1109/ICECET58911.2023.10389377>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.