Project Two

Dennis T Sherpa

Southern New Hampshire University

CS 320

Instructor Federico Bermudez

December 13th, 2024

Project Two

In this paper, I will provide a summary and reflection report explaining how I analyzed various approaches to software testing based on requirements and applied appropriate testing strategies to meet those requirements while developing the customer's mobile application. This report will be based on my experience completing Project One.

1. Summary

I will explain my unit testing approach for the three features of the mobile application (Contact Service, Task Service, and Appointment Service) by addressing the questions proposed:

- To what extent did my testing approach align with software requirements?
    - Contact Service
        - My testing approach aligned entirely with the Contact Class software requirements because, in general, I tested every aspect of creating and updating a contact.
        - My testing approach aligned entirely with the Contact Service Class software requirements because, in general, I tested the ability to add, delete, and update a contact.
    - Task Service
        - My testing approach aligned entirely with the Task Class software requirements because, in general, I tested every aspect of creating and updating a task.

        - My testing approach aligned entirely with the Task Service Class software requirements because, in general, I tested the ability to add, delete, and update a task.
    - Appointment Service
        - My testing approach aligned entirely with the Appointment Class software requirements because, in general, I tested every aspect of creating and updating an appointment.
        - My testing approach aligned entirely with the Appointment Service Class software requirements because, in general, I tested the ability to add and delete an appointment.

- Based on coverage percentage, how do I know my JUnit tests were effective?
    - I implemented EclEmma in Eclipse to obtain a coverage report for my Project One submission, which contained all three required features (Contact Service, Task Service, and Appointment Service):
        - Project One
            - Coverage Report: Project One Coverage.zip
            - With a coverage percentage of 85%, my JUnit tests effectively ensured that almost all of my Project One code–including Contact, Task, and Appointment Service–was executed during testing.
            - This high level of coverage demonstrates that my JUnit tests efficiently tested key functionalities and met the Test

Coverage requirement of Exemplary (100%) noted in the Project One rubric.

- How did I ensure that my code was technically sound?
    - I ensured my code was technically sound by designing each test to validate a single requirement with explicit preconditions, actions, and expected outcomes.
    - An example that supports my above statement is the following code that tests when a task is created with a task ID that exceeds ten characters:

```
@Test
public void testWhenTaskIdExceedsCharacterLimit() {
    assertThrows(IllegalArgumentException.class, () -> {
        new Task("01234567890123456789", "Testing
        Task", "This a task created for JUnit testing");
    });
}
```

    - This test ensures that the Task Class requirement, no task ID being more than ten characters, is met.
    - The rest of the tests in my Project One submission are similar to the above code/test, concentrating on testing a single requirement.
- How did I ensure my code was efficient?
    - I ensured my code was efficient by minimizing redundancy, focusing on a single functionality, anticipating potential failures, and not incorporating anything unnecessary.

- An example that supports my above statement is the following code that executes before each test case, ensuring that each test starts with the same initial conditions without duplicating the initialization code in every test.

```
@BeforeEach public void setUp() {
        taskService = new TaskService();

        taskOne = new Task("0123456789", "Testing Task 1",
        "This is the 1st task created for JUnit testing");
        taskTwo = new Task("9876543210", "Testing Task 2",
        "This is the 2nd task created for JUnit testing");
        taskThree = new Task("13579", "Testing Task 3", "(This is
        the 3rd task created for JUnit testing");
    }
```

- This test ensures that the required objects are initialized before each test in the Task Service Class, allocating resources efficiently and reducing redundancy.
- The rest of the tests in my Project One submission are similar to the above code/test, maximizing efficiency in their respective areas.

2. Reflection

I will start by describing the characteristics of software testing techniques I employed in Project One and the characteristics of software techniques I did not employ. Then, I will explain the practical uses and implications of each technique–that I did and did not use–for different

software development projects and situations. Lastly, I will work on explaining my mindset while completing Project One.

a. Testing Techniques

The descriptions of software testing techniques that I employed for Project One are as follows (GeeksforGeeks. (2024)):

1. Functional testing: Tests the functional requirements of the software to ensure they are met.
    a. Specific detail: I employed JUnit tests to ensure each feature met its requirements. For example, in my Project One submission (Feature: Task Service), I employed a JUnit test in TaskTest.java to ensure that in Task.java, a task cannot be created with a task ID exceeding ten characters. The following is the code:

```
// Test when a task is created with a task ID that exceeds ten characters
@Test
public void testWhenTaskIdExceedsCharacterLimit() {
        assertThrows(IllegalArgumentException.class, () -> {
                new Task("01234567890123456789", "Testing Task", "This a task created for JUnit testing");
        });
}
```

    b. This ensured that one of the functional requirements, a task's task ID not exceeding ten characters, of Project One was met.

c. I ensured each functional requirement of each feature through similar JUnit tests as above.

2. Unit Testing: Tests the individual units or components of the software to ensure they are functioning as intended.

    a. Specific detail: I thoroughly and repetitively refined my code for the JUnit tests to ensure they functioned as intended. For example, in my Project One submission (Feature: Appointment Service), I had to refine my code in AppointmentTest.java several times to succinctly test that an appointment with a date in the past cannot be created. The following is the final code from the refinements:

```
// Test when an appointment is created with an appointment date in the past
(specifically, two days before today)
@Test
public void testWhenAppointmentDateIsInThePast() {
        assertThrows(IllegalArgumentException.class, () -> {
                new Appointment("0123456789", new
                Date(System.currentTimeMillis() - 2L * 24 * 60 * 60 * 1000), "This
                is an appointment created for JUnit testing");
        });
}
```

    b. This took significant time because I had to figure out how to utilize the Date library to create a Date object that held a past date.

3. System Testing: Tests the complete software system to ensure it meets the specified requirements.
    a. After completing and testing my JUnit tests, I ensured all features in my Project One submission met their respective requirements by checking that each feature's JUnit tests passed and no errors arose in the code. For example, in my Project One submission (Feature: Appointment Service), I ensured through my JUnit test in AppointmentTest.java that an appointment could be created in Appointment.java. The code is as follows:

```
// Test when an appointment creation succeeds
@Test
public void testAppointmentCreationSucceeds() {
    Appointment appointment = new Appointment("0123456789", this.date,
    "This is an appointment created for JUnit testing");
    assertEquals("0123456789", appointment.getAppointmentId());
    assertEquals(this.date, appointment.getAppointmentDate());
    assertEquals("This is an appointment created for JUnit testing",
    appointment.getAppointmentDescription());
}
```

The software testing techniques that I did not employ in Project One are as follows (GeeksforGeeks. (2024)):

1. Security Testing: Tests the software to identify vulnerabilities and ensure it meets security requirements.

    a. Identifies weaknesses in authentication, authorization, and data protection.
    b. Ensures secure communication channels and proper encryption.
    c. Verifies compliance with security regulations.

2. Exploratory Testing: A type of testing where the tester actively explores the software to find defects without following a specific test plan.
    a. Testers use their understanding of the system to identify unexpected behaviors or defects.
    b. Emphasizes learning, investigation, and adaptability during the testing process.
    c. Typically used when requirements are incomplete or during early testing phases.
3. Usability Testing: Tests the software to evaluate its user-friendliness and ease of use.
    a. Evaluate the system's ease of use.
    b. Identifies issues that lengthen a user's learning curve.
    c. Ensures software meets user expectations regarding navigation, design, and functionality.

The practical uses and implications of each software technique I discussed for different software development projects and situations are as follows (GeeksforGeeks. (2024)):

- Functional Testing
    - Practical Uses

        - Ensures that the software meets the specified functional requirements.
        - Verifies critical functional features of the software.
    - Implications
        - Enhances user satisfaction by providing expected software functions.
        - Helps deliver software that meets stakeholders' and developers' intended vision.
- Unit Testing
    - Practical Uses
        - Validates the correctness of individual units or components of the software.
        - Essential for projects built through agile development cycles.
    - Implications
        - Reduces cost and time by finding errors early in development.
        - Improves code quality and readability.
- System Testing
    - Practical Uses
        - Tests the entire system to ensure all components work together seamlessly.
        - Vital for overall project success.
    - Implications
        - Detects issues that impact the system.

        - Ensures the system meets business requirements.

- Security Testing
    - Practical Uses
        - Identifies vulnerabilities and ensures the system is protected from cyber threats.
        - Strengthens software handling sensitive data.
    - Implications
        - Enhances user trust in the security of the software.
        - Safeguards user's data against various threats.
- Exploratory Testing
    - Practical Uses
        - It helps uncover hidden or unexpected defects.
        - Useful in rapidly evolving projects.
    - Implications
        - Identifies edge cases and previously unknown issues.
        - Encourages creative problem-solving, which may lead to revolutionary software.
- Usability Testing
    - Practical Uses
        - Evaluates the user-friendliness of a software's user interface.
        - Ensures positive user experiences for interactive software.
    - Implications
        - Improves user satisfaction and software usage.

- Provides insights to compete with today's highly interactive software.

b. Mindset

i. Caution

While working on Project One, I adopted a mindset that embodied attention to detail. This meant, in general, ensuring that all requirements were met and tested to achieve a coverage of 80% or higher. My approach required the employment of caution, which drove me to develop concise code, implement precise argument exceptions, coordinate between classes, incorporate sound logic, and create an error-free program. Furthermore, to successfully implement caution, it was important to appreciate the complexity and interrelationships of the code I was testing. Understanding the nuance and connections between methods, classes, and files was key to meeting the requirements.

In the following, I will provide an example (with/ two sub-examples) that illustrates the caution I employed in my Project One submission. First, in my AppointmentService.java file, I incorporated the following code:

```
// Add a new appointment with a unique appointment ID
public boolean addAppointment(Appointment appointment) {
    // If an appointment with the appointmentId of the parameter
    appointment already exists
    if (appointments.containsKey(appointment.getAppointmentId())) {
        // Do not add appointment
```

```
            return false;
        }
        // Add a new appointment to the list of appointments
        appointments.put(appointment.getAppointmentId(), appointment);
        return true; // Appointment added
    }
```

Second, in my AppointmentServiceTest.java file, I incorporated the following code that tests the above code:

```
    // Test the appointment service of adding appointments
    @Test
    public void testAddAppointment() {
        // Test that three appointments, each with a unique appointment ID,
        can be added to the list of appointments
        assertTrue(appointmentService.addAppointment(appointmentOne)
        );
        assertTrue(appointmentService.addAppointment(appointmentTwo
        ));
        assertTrue(appointmentService.addAppointment(appointmentThre
        e));

        // Test that a pre-existing appointment cannot be added
        assertFalse(appointmentService.addAppointment(appointmentOne
        ));
```

```
}
```

In general, this example illustrates the following:

1. I created a testable method that met one of the project requirements for adding an appointment.
2. I created a JUnit test to test the method for adding an appointment, ensuring that a valid appointment could be added and an invalid appointment could not.

ii. Bias

The ways I tried to limit bias in my review of the code are as follows:

1. Developing code that met the requirements of Project One–nothing more or less.
2. Removing all errors and unnecessary code in the program objectively.
3. Constantly reviewing and refining code according to the requirements open-mindedly.

On the software developer side, bias could be a concern if I were responsible for testing my own code. Like any other human, I have natural negative tendencies to support my ego, feel righteous, be lazy, and ignore mistakes. These tendencies are avoidable, and I will always aim to overcome them; however, they remain a concern.

iii. Discipline

The importance of being disciplined in my commitment to quality as a software engineering professional is imperative. I truly believe that a software engineer cannot achieve the idealistic level of quality for their program without

discipline. As I aspire to become a software engineering professional, I understand my commitment to quality must be unwavering, and that can only be achieved with foundational discipline.

It is important not to cut corners when writing or testing code because only then can the software engineer realistically hope their code makes a meaningful impact. The competition in today's world of technology is extremely high. Even when the best software engineers don't cut corners, they fail to achieve the dreams they hold for their programs. Therefore, to even have a chance to succeed, a software engineer must start by not cutting corners.

To avoid technical debt as a practitioner in the field, I plan to gain a deep understanding of what I want to provide/solve, create a roadmap to success (while not spending excessive time), code and test early, follow the best coding practices, balance speed and quality, eliminate (or at least try to eliminate) bias, and constantly review and revise. By achieving these principles, I believe I can avoid technical debt.

Citations

GeeksforGeeks. (2024). *Software testing techniques*. https://www.geeksforgeeks.org/software-testing-techniques/