**Summary and Reflection**

Ikechukwu Onuoha

Southern New Hampshire University

CS-320-J8975 Software Test Automation& QA 23EW1

Robert Tuft

October 15, 2023

**PART ONE: SUMMARY**

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

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

My approach was aligned with the software requirements extensively. To achieve this, I created the `ContactTest` class to cover various aspects of the `Contact` class. I used test methods that focused on creating valid contacts, handling invalid inputs, and checking constraints. The test method created a valid contact by providing all the required fields which were contactID, firstName, lastName, phone, and address with values that fell within the specified constraints like the maximum length for firstName, lastName, and address. I then used assertions to verify that the contact object's attributes matched the expected values. This ensured that the contact was created correctly.

Example: In the `testCreateValidContact` method, assert is used to confirm a valid contact is created and its properties match the expected values.

For the Task service, the testTaskCreationWithValidData method created a task with valid data and asserted that it was created successfully with the expected attributes. The testTaskNameUpdate method tested the ability to update the task's name and verified that the update was successful. The testTaskCreationWithInvalidTaskId method tested the case where an invalid task ID was used during task creation, resulting in an expected exception. The testTaskCreationWithInvalidName method tested the case where an invalid name that was too long was used during task creation, also resulting in an expected exception. The testTaskCreationWithNullDescription method tested the case where a task was created with a null description, which is allowed.

Example: In the `testUpdateTaskDescription`, I ensure that the task description is updated correctly and matches the expected value.

For the Appointment Service, the @Before annotated setUp method was used to initialize the appointmentService before each test method was executed, ensuring a clean state for testing. The testAddAppointment method created a future date, added an appointment with valid data, and asserted that the appointment was added successfully. The testDeleteAppointment method added an appointment and then deleted it, verifying that the appointment was deleted successfully. The testAddDuplicateAppointmentID method added two appointments with the same ID, resulting in an expected exception. The testAddPastAppointmentDate method attempted to add an appointment with a past date, leading to an expected exception. The testAddInvalidDescription method tried to add an appointment with an invalid description that was too long, causing an expected exception. The testDeleteNonexistentAppointment method tried to delete an appointment with a non-existent ID, resulting in an expected exception.

Example: In `testAddDuplicateAppointmentID`, I verified that the code throws an `IllegalArgumentException` when adding a duplicate appointment.

**ii. 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?**

Based on the coverage percentage, my Junit tests were effective because I ensured valid input, test isolation, and efficient error handling. In the ContactTest class, there was a test method that covered valid contact creation.

@Test

public void testValidContactCreation() {

Contact contact = new Contact("ID12345", "John", "Doe", "1234567890", "123 Main St");

assertEquals("ID12345", contact.getContactID());

assertEquals("John", contact.getFirstName());

assertEquals("Doe", contact.getLastName());

assertEquals("1234567890", contact.getPhone());

assertEquals("123 Main St", contact.getAddress());

}

This test ensured that when I created a contact with valid data, the properties were correctly set.

In the `AppointmentServiceTest`, I included tests for exceptions like `testAddDuplicateAppointmentID` and `testAddPastAppointmentDate`.

For test isolation, there were isolated test methods for different functionality, such as adding tasks, deleting tasks, and updating task properties.

@Test

public void testAddTask() { // Test for adding a task

}

@Test

public void testDeleteTask() { // Test for deleting a task

}

@Test

public void testUpdateTaskName() { // Test for updating task name

}

Each test method in this class was independent and did not rely on the outcome of other tests.

Lastly, for error handling, in the AppointmentServiceTest class, there were tests for error handling, such as detecting duplicate appointments.

@Test(expected = IllegalArgumentException.class)

public void testAddDuplicateAppointmentID() {

Date futureDate = new Date(System.currentTimeMillis() + 86400000); // 24 hours ahead

appointmentService.addAppointment("Appt3", futureDate, "Test appointment 3");

appointmentService.addAppointment("Appt3", futureDate, "Duplicate appointment");

}

This test ensured that the application correctly handled situations where duplicate appointments were attempted.

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

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

Writing JUnit tests for this project was a valuable experience. It required careful consideration of different scenarios and edge cases. Before writing tests, there was the need to first understand the code under test which was crucial in designing effective tests. Designing test cases that covered various scenarios to ensure comprehensive coverage of the code followed before execution of the tests regularly to verify that they passed under normal and edge conditions. Failures were also analyzed and addressed promptly.

I ensured the technical soundness of my code by rigorously testing different code paths. In the `AppointmentServiceTest`, validation ensured that the code correctly handled exceptions in methods like `addAppointment` and `deleteAppointment`. This approach guaranteed that the software behaved as intended and responded appropriately to exceptional situations.

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

I ensured code efficiency by designing tests that executed quickly and without unnecessary overhead. In the `ContactTest`, I tested contact creation and validation efficiently, ensuring that the code performed optimally during these operations. I also used assertEquals to verify that the created contact object's properties matched the expected values. This assertion method is concise and performs efficiently.

**PART TWO: REFLECTION**

**a. Testing Techniques**

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

I used various testing techniques in this project. The first one was unit testing which involves testing individual units or components of code in isolation to ensure their correctness (Hamilton, 2022). The testValidContactCreation method focused on testing the Contact class's ability to create a valid contact object with specific properties. This test method was isolated and did not depend on other parts of the codebase. It checked if the created Contact object contained the expected values for its properties. Unit testing helped catch and rectify issues at an early stage, ensuring that each component behaved as expected.

Secondly, I also used Black-box testing which focused on testing the functionality of a component without considering its internal implementation details.

Black-box testing was used to test the addAppointment method of the AppointmentService. The test focused on the external interface and expected behavior of the AppointmentService. It adds an appointment to the service with a future date and checks whether the appointment is correctly added to the list of appointments. Black-box testing ensured that the software met specified requirements and behaved as users would expect.

I also used Assertions that verified that the actual output or behavior of the code matched the expected results. Assertions using `assertEquals` and `assertNotNull` helped validate the correctness of code execution. In `TaskServiceTest`, assertions were put to check if task properties matched the expected values. Assertions provide a clear way to determine whether the code behaves as intended and help identify discrepancies (JUnit - API, n.d.).

Lastly, I used Exception handling testing which ensured that the code correctly threw appropriate exceptions when encountering errors or exceptional conditions. In the `AppointmentServiceTest`, I tested scenarios where exceptions should be thrown, such as adding a duplicate appointment or adding an appointment with an invalid date. Exception handling testing guaranteed the code responded appropriately to error conditions, enhancing its robustness.

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

Other testing techniques that I did not use include smoke testing, also known as "build verification testing," which involves quickly checking if a software build is stable after each build (Boni García, 2017). It was not employed in this project, as it is typically used in larger projects with frequent builds to identify showstopper defects early. Smoke testing is valuable for quickly rejecting unstable builds, saving time and effort in comprehensive testing.

Performance testing is another technique that assesses how a system performs under various conditions, including load, stress, and scalability. This technique was not used in this project but is crucial for evaluating system performance, identifying bottlenecks, and optimizing performance. For example, in an e-commerce application, performance testing would simulate a high number of concurrent users to ensure the system handles load efficiently.

Security testing, another technique, focuses on identifying vulnerabilities and ensuring the security of sensitive data (Boni García, 2017). It was not employed in this project, but it is essential for projects dealing with confidential or regulated data. For example, in a healthcare application, security testing would assess vulnerability to external attacks and verify data encryption.

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

1. Unit Testing

Practical Uses: Unit testing is used in projects especially those using agile methodologies. It is essential to ensure that individual components or units of code work correctly in isolation (Hamilton, 2022). In a web application, unit testing can verify that functions responsible for user authentication, data retrieval, or input validation behave as expected.

Implications: According to Hamilton (2022), unit testing helps catch and fix bugs early in the development cycle, reducing the cost of fixing issues later. It provides a safety net for code changes and encourages developers to write modular and maintainable code.

2. Black-Box Testing

Practical Uses: Black-box testing is suitable for projects where the focus is on verifying whether the software meets specific requirements or user expectations. It is commonly used in user interface testing, integration testing, and system testing. In an e-commerce website, black-box testing can validate the functionality of the shopping cart, checkout process, and search functionality from a user's perspective.

Implications: Black-box testing ensures that the software behaves as intended, regardless of its internal implementation. According to Boni García (2017), it helps identify discrepancies between expected behavior and actual behavior and validates that the software meets functional requirements.

3. Assertions

Practical Uses: Assertions are a fundamental part of testing in various software projects. They are used to verify correctness and ensure that the code produces the expected output or state (JUnit - API, n.d.). Assertions are applicable in unit testing, integration testing, and system testing. For example, in a financial application, assertions can be used to validate that financial calculations are accurate and produce the expected results.

Implications: Assertions provide a means to automatically check code correctness. They serve as documentation for expected behavior, making it easier for developers to understand code requirements and enabling faster bug detection and resolution.

4. Exception Handling

Practical Uses: Exception handling testing is essential in projects where robust error handling is critical, such as financial software or safety-critical systems. It ensures that the software responds appropriately to exceptional conditions and errors, maintaining data integrity and user safety (Sinha & Harrold, 2000). For example, in medical device software, exception handling testing can confirm that the system handles hardware failures gracefully.

Implications: Exception handling testing helps identify situations where the code may fail, and it verifies that the software handles these situations without crashing or causing data corruption. It is crucial for maintaining system reliability and user trust.

5. Smoke Testing

Practical Uses: This is particularly useful in larger projects with frequent builds, ensuring that basic functionality is not broken after each build. It helps save time by quickly rejecting unstable builds, making it suitable for continuous integration environments.

Implications: Errors are identified at an early stage, which tends to reduce the cost and time of fixing them.

6. Performance Testing

Practical Uses: Vital for projects where system performance is critical, such as online gaming platforms or high-traffic websites. It helps identify performance bottlenecks and ensures that the software can handle expected loads (Boni García, 2017).

Implications: Performance tests can reveal unpredictable behavior under stress which might not be evident during regular functional tests.

7. Security Testing

Practical Uses: Essential for projects dealing with sensitive data or where security vulnerabilities could have severe consequences. It helps identify and address security weaknesses and is commonly used in financial, healthcare, and government applications.

Implications: Security is not a one-time activity. New vulnerabilities emerge regularly, so continuous security testing and updates are necessary.

**b. Mindset**

**i. 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.**

While working on testing codes, I adopted a good mindset, emphasizing caution and thoroughness. Testing is a critical part of software development, and any negligence could lead to severe consequences (Hambling et al., 2019). I was very cautious when working on this project by ensuring that task IDs in the task code were validated appropriately. This validation helped maintain data consistency. Tasks with invalid IDs would not be added, preventing issues related to data retrieval or updates. I also enforced length limits on task names and descriptions, similar to the appointment code. This limitation was essential to prevent excessive data and maintain readability and user-friendliness.

Understanding the complexity and interrelationships of the code was vital for effective testing in all codes. In the TaskService, understanding how tasks were added and updated was very essential. Testing one function's correctness requires understanding its interactions with other parts of the code. In the ContactService, comprehending how contacts were added, updated, and deleted was necessary for thorough testing.

**ii. 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.**

I tried to limit bias in my review of the code by ensuring consistency in standards, whereby I followed consistent coding standards and best practices throughout the codebase. In the contact app, I ensured that variable naming conventions, indentation, and code organization were uniform. This consistency helped in avoiding personal biases related to code style preferences. I also concentrated on evaluating the functionality of the code rather than personal preferences. In the appointment service, I focused on whether the code met the specified requirements and provided the expected functionality.

I also ensured that the code was well-documented with comments explaining complex logic or design decisions. This transparency reduces the potential for misinterpretation and bias when others review the code. For instance, in the task app, I added comments to clarify why specific validation rules were in place. Lastly, I wrote test cases before implementing the code. This approach focused on objective validation of code functionality based on predefined requirements, reducing the chance of subjective bias affecting the assessment. In the task service, I wrote tests to validate task creation and updates.

I do believe that bias in self-testing is indeed a concern. According to Calikli et al. (2010), when testing one's own code, there might be a tendency to overlook potential issues or test cases that challenge the code's correctness (confirmation bias). For example, if I'm testing my own code for the contact app and I'm confident in its correctness, I might unconsciously skip certain test scenarios or not dig deep enough into potential edge cases. Additionally, when testing one's code, there might be an inclination to blame external factors for failures rather than accepting responsibility (self-attribution bias). For example, if a test case fails during self-testing, I might initially think it's due to an issue with the testing environment or data, rather than considering that there could be a flaw in my code. To add to this, overconfidence bias can lead to a lack of thoroughness in testing, assuming that the code is free from errors. For example, as a developer, I might believe that my code is so well-written that extensive testing isn't necessary, potentially missing critical issues.

**iii. 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.**

Technical debt is like accumulating interest on a loan; the longer it's left unaddressed, the more it grows. Cutting corners in code quality and testing accumulates this debt, making future development slower, riskier, and more error-prone. By adhering to coding and testing standards, we reduce the interest on this debt and ensure a smoother development process (Krishna & Basu, 2012). Inadequate code quality and testing can directly impact end-users. For example, in the contact management application, if I cut corners and didn't validate inputs properly, users could experience data corruption or loss. This would lead to user dissatisfaction and could result in a loss of trust in the software. Skipping testing, as mentioned earlier, can lead to undetected bugs and system instability. For instance, in the appointment service, if I didn't thoroughly test the handling of edge cases like duplicate appointments or invalid dates, users might experience unexpected crashes or erroneous behavior, damaging the software's reputation.

To avoid technical debt as a practitioner, adherence to coding guidelines and best practices should be a non-negotiable rule, something I plan to always adhere to. For instance, in the task service code, consistently validating inputs and adhering to naming conventions ensures code is maintainable and free from vulnerabilities. I will also regularly review and refactor code to ensure it remains clean and efficient. In the contact service, periodically revisiting and optimizing data storage and retrieval methods could help avoid performance issues. Additionally, I will always prioritize testing given that comprehensive testing should be an integral part of the development process. In the appointment service, extensive testing of boundary conditions, error handling, and performance under different loads is essential to maintain code quality and reliability. Lastly, given the rapid evolution of the software development field, staying updated with industry trends and best practices is essential. For instance, in the contact app, keeping up with changes in data encryption and security practices ensures that sensitive user data remains protected.

References

Boni García. (2017). *Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications*. Packt Publishing, Birmingham (Gran Bretaña.

Calikli, G., Aslan, B., & Bener, A. (2010, September 1). *Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size, Experience and Reasoning Skills*. Oro.open.ac.uk. https://oro.open.ac.uk/45370/#:~:text=However%2C%20software%20developers%20and%20testers

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *SOFTWARE TESTING : an istqb-bcs certified tester foundation guide.* Bcs, The Chartered Instit.

Hamilton, D. (2022, December 6). *JUnit Tutorial With Examples: Setting Up, Writing, and Running Java Unit Tests*. Parasoft. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/#What\_Is\_Unit\_Testing

*JUnit - API*. (n.d.). Www.tutorialspoint.com. Retrieved October 13, 2023, from https://www.tutorialspoint.com/junit/junit\_api.htm

Krishna, V., & Basu, A. (2012). Minimizing technical debt: developer’s viewpoint. *International Conference on Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012)*. https://doi.org/10.1049/ic.2012.0147

Sinha, S., & Harrold, M. J. (2000). Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, *26*(9), 849–871. https://doi.org/10.1109/32.877846