**Project Two**

Department of Computer Science and Engineering, Southern New Hampshire University

CS-320 Software Test Automation & QA

Tanisha D. Jacks

June 18, 2023

**Summary**

**Unit testing approach**

The project encompasses unit tests for six essential features: the Appointment class, the AppointmentService class, the Contact Class, the ContactService class, the Task class, and the TaskService class. These unit tests serve to cover a wide range of scenarios and effectively validate the behavior of the code. Notably, the three classes, Appointment, Contact, and Task, share similar fields and behavior, thus employing an identical testing approach for these classes ensures consistent and thorough testing coverage. Similarly, the other three classes, AppointmentService, ContactService, and TaskService, being identical services, follow the same testing approach. By adopting a unified testing approach for classes with similar structures and behavior, the project ensures efficient and effective testing while reducing duplication of efforts. This approach enhances maintainability and facilitates easier future modifications and enhancements to the system.

For the Appointment, Contact and Task classes, the unit tests cover both valid and invalid inputs for the constructor and the mutator methods. They test cases such as valid arguments, invalid appointment ID (too long or null), past appointment date, null appointment date, invalid description (too long or null), and more. Each test case verifies that the appropriate exception is thrown when necessary and that the state of the Appointment object is correctly updated when valid arguments are provided.

For the AppointmentService, ContactService and TaskService classes, the unit tests cover the add, delete and update methods (except for the AppointmentService class which does not include update method). The tests ensure that appointments, contacts or tasks are added correctly to the list and that an exception is thrown when trying to add an appointment, contact or task with an existing ID. The deletion of an item is also tested, ensuring that the item is removed from the list and that an exception is thrown when trying to delete a non-existing one.

The unit tests align with the software requirements by validating the behavior of the code against the expected outcomes. They cover various edge cases and scenarios that the code should handle correctly. The tests verify that the code adheres to the requirements related to ID uniqueness, appointment date validation, length limitations of first name, last name, address, description or other fields.

To assess the overall quality of the JUnit tests, it is crucial to consider both the coverage percentage and the range of test cases included. By executing the code with the "Coverage As JUnit Test" option in Eclipse, the coverage percentage was determined to be 100%. This implies that every line of code within the tested classes was executed at least once during the testing process, indicating a comprehensive coverage of the codebase.

**Writing JUnit tests experience**

To ensure that the code is technically sound, test setup and teardown, as well as the test assertions practices were adopted.

The use of @BeforeEach and @AfterEach annotations ensures that the necessary setup and teardown actions are performed before and after each test case. This helps create a clean and consistent environment for testing. For example, in the AppointmentTest class, the @BeforeEach method is used to set up a valid appointment object:

| @BeforeEach void setUp() {  currentDate = new Date()*;*  appointmentId = "A123456789"*;*  appointmentDate = currentDate*;*  description = "Periodontal Disease treatment appointment"*;*  appointment = new Appointment(appointmentId, appointmentDate, description)*;* } |
| --- |

By setting up the required objects and data before each test, I ensure that the tests are independent and reliable.

In addition, the use of assertions helps verify the expected behavior of the code being tested. Specific lines of code within the tests demonstrate this. For example, in the AppointmentTest class, the following line asserts the equality of the appointment ID:

| assertEquals(appointmentId, appointment.getAppointmentId())*;* |
| --- |

This line of code ensures that the appointment ID retrieved from the appointment object is equal to the expected appointment ID. Such assertions are used throughout the tests to validate the correctness of the code.

Furthermore, to ensure efficient and effective test code, several practices are followed. The focus is on writing tests that provide targeted coverage without unnecessary duplication or complexity. For example, the Task class has separate test methods for validating task ID length, name length, description length, and updating task properties. Each test covers a specific aspect, avoiding redundancy.

**Reflection**

**Testing Techniques**

The project utilized various software testing techniques, including unit testing, black box testing, boundary value analysis, and error handling testing. Unit tests were written using JUnit, focusing on individual units of code like methods and constructors. Black box testing involved test cases that validated expected behavior based on inputs and outputs, ensuring correct appointment addition, retrieval, and updates. Boundary value analysis ensured edge case handling, such as maximum length of ID, first name, last name, address, description and so on, as well as behavior with current or past dates. Additionally, the test cases verified the code's ability to throw specific exceptions for invalid inputs like IDs or descriptions. These techniques collectively ensure thorough testing and identification of potential issues in the code.

While the project employed various software testing techniques, there were other techniques that were not utilized but could have improved the software's quality. Performance testing, for instance, could have been implemented to assess the code's performance and scalability under different workloads or stress conditions, identifying potential bottlenecks and resource issues. Load testing, stress testing, and endurance testing are examples of techniques that could have been used to evaluate performance characteristics. Additionally, if the code handles sensitive data or has security implications, security testing would have been valuable to identify vulnerabilities and ensure robustness against attacks. Techniques such as penetration testing, vulnerability scanning, and threat modeling could have been employed to assess the code's security posture. These additional techniques would have further enhanced the quality and reliability of the software.

Testing techniques have practical uses and implications for various software development projects and situations. Unit testing, performed by developers during the development phase, ensures the correct functioning and adherence to specifications of each unit. It helps identify bugs early, simplifies debugging, and reduces issues in later stages. Black box testing examines the behavior of a system without knowledge of its internal structure, validating functionality from a user's perspective and identifying discrepancies. Boundary Value Analysis (BVA) focuses on identifying errors occurring at input value boundaries, crucial for critical applications where mishandling boundary conditions can have severe consequences. Error handling testing evaluates how well software handles unexpected or erroneous conditions, verifying the effectiveness of error messages, exception handling, and recovery mechanisms. These testing techniques are valuable for different types of software projects, ensuring correctness, meeting requirements, and enhancing software quality.

**Mindset**

***Caution***

As a software tester, I understood the importance of thoroughly testing the code and identifying potential issues or bugs. I employed caution by carefully designing test cases to cover different scenarios and boundary conditions. Besides, understanding how different parts of the code interacted with each other also helped me identify potential areas of impact and ensure comprehensive test coverage.

For example, in the Appointment class, I validated the inputs in the constructor to ensure that the provided arguments met the specified requirements. I employed caution by checking for null values, enforcing length limits, and validating the appointment date to prevent invalid or inconsistent data. In the AppointmentService class, I carefully considered the interaction between appointments and the unique appointment ID requirement. When adding a new appointment, I checked for the existence of an appointment with the same ID to avoid duplicate entries. This interrelationship between appointments and their IDs was critical to maintaining data integrity.

Additionally, in the AppointmentServiceTest class, I thoroughly tested the addAppointment() and deleteAppointment() methods. I employed caution by adding not one but multiple appointments with different IDs and dates to verify the correctness of the functionality. I also checked for exceptions when attempting to add an appointment with an existing ID or delete a non-existent appointment.

***Bias***

To minimize bias in the code, I follow certain practices. First of all, I tried to focus on the code itself. Specifically, I focused on evaluating the code based on its adherence to best practices, readability, maintainability, and correctness, rather than making subjective judgments about personal coding preferences.

Secondly, I made sure that I test both valid and invalid scenarios. The test cases cover various scenarios, including both valid and invalid inputs. This approach ensures that the code is tested under different conditions and can handle both correct and incorrect data. For example, the tests testIdTooLong and testIdNull check for the validation of the appointment ID, while testPastDate and testDateNull test the validation of appointment dates.

Finally, boundary value testing was implemented. For example, the test testDescriptionTooLong checks if the code correctly handles a description that exceeds the maximum character limit of 50. Implementing boundary value testing can help avoid bias in writing test code because it focuses on testing the extreme values and boundaries of input ranges. Bias in test code can occur when only typical or average values are considered, potentially overlooking edge cases and boundary conditions that may lead to bugs or unexpected behavior.

***Discipline***

Being disciplined and committed to quality as a software engineering professional is extremely important as cutting corners while writing test code can have severe consequences and lead to technical debt. Writing high-quality code and conducting thorough testing ensures the reliability and robustness of the software. In this project, I achieve this by covering different scenarios, such as valid and invalid inputs and asserting exceptions. Testing for expected exceptions, such as an IllegalArgumentException for invalid inputs, helps ensure that the code behaves as intended in error conditions.

Cutting corners in code quality can make software harder to understand, maintain, and extend over time. Technical debt accumulates when suboptimal design choices and shortcuts are made. To avoid technical debt, I made sure to follow best practices and maintain a disciplined approach to code quality. For instance, in this project, I utilized clear and descriptive test names to improve readability and make it easier for other developers to understand the purpose and behavior of the tests.

Last but not least, cutting corners in code quality can alsoresult in a poor user experience. Bugs, crashes, and inconsistencies can frustrate users and harm the reputation of the software or the organization behind it. Prioritizing quality ensures that users have a positive experience and builds trust.