SNHU: CS-320

Software Test Automation & QA

Module 7 – Project 2

Matthew Bandyk

When completing the development of the mobile application, the requirements provided were the driving force in the development. All services developed, which included the Contact, Task, and Appointment services, were developed following the requirements specifically. Each service required specific variables to be available which needed to follow specific guidelines. To ensure those requirements were met, variables were created, and input validation was initiated through the classes to accomplish this.

For example, in the task service, there was a requirement to have a variable to capture the name of the task for each task object which could not be null and had to be a length of 20 or less. In the class, I created a validation in the task object creation method that confirmed that the passed name data was in fact not null and not greater than 20. This can be seen on lines 31 through 33 in the Task.java file. This approach was utilized in all the service files based on all provided requirements.

Additionally, other requirements included the ability to remove existing objects for each of the classes, as well as the ability to modify existing objects. I’ve created the methods that support these requirements while also validating the inputs to ensure they continue to adhere to the other requirements provided.

After completing the development of the services, I also implemented JUnit tests to validate that the system would operate as intended and within the provided requirements. To ensure that proper testing was completed, coverage of over 95% was achieved for each of the services. By aiming for a coverage percentage close to 100%, we can confidently assert that the system undergoes thorough checks and validates the requirements. It is important to strive for maximum coverage when developing test cases. However, it is essential to note that achieving high coverage does not guarantee that every possible scenario is reviewed. For instance, one requirement specifies that the phone number in the Contact Class should be exactly 10 digits long. While a test case can be created to capture an incorrect input like "123," it may not address situations where the input exceeds 10 digits. To account for these scenarios, I have developed additional test cases that cover such situations.

To ensure that the code I developed for the 3 services was technically sound, I utilized industry best practices and effectively utilized the assertion capabilities provided by the JUnit framework. For instance, in the TaskServiceTests class, I incorporated the @BeforeEach annotation, which facilitates the creation of a new TaskService object for each individual test case. By strategically adding comments throughout the test classes, I enabled users to easily understand the code's functionality and purpose. Moreover, I maintained consistent naming conventions across all test cases, promoting readability and maintainability. As an illustration, when conducting multiple test cases for the updateTask method in the TaskServiceTests JUnit class, I employed standardized naming conventions. For instance, I used the testUpdateTaskName() format for all tests associated with updating the task's name, while appending additional words at the end of each name to indicate the specific aspect being tested, such as null or length. Examples from the code base are below.

public void testUpdateTaskNameNull()

public void testUpdateTaskNameLength()

Additionally, I developed efficient code by ensuring that it was simplistic in design, fast to execute, and repeatable to reduce workload. Multiple tests were developed for individual methods to test the different possible inputs, exceptions, and outputs. This ensured I was able to account for different situations while validating the system would perform as expected, while increasing my ability to identify, and then solve, bugs that could arise. Examples of this can be found throughout my code base, for example, when looking at the test cases for the setNumber method in the Contact Class, I developed three test cases to ensure exceptions were thrown if the input was incorrect.

// Test to validate if creating a contact throws exception if number is null

@Test

**public** **void** testNumberNull() {

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Contact("CS 305", "John", "Doe", **null**, "123 Main St");

});

}

// Test to validate if creating a contact throws exception if number is not 10 in length (Shorter)

@Test

**public** **void** testNumberLengthToShort() {

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Contact("CS 305", "John", "Doe", "123", "123 Main St");

});

}

// Test to validate if creating a contact throws exception if number is not 10 in length (Longer)

@Test

**public** **void** testNumberLengthToLong() {

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Contact("CS 305", "John", "Doe", "9876543210123456789", "123 Main St");

});

}

During the testing of the three services I developed, I primarily utilized unit testing techniques, also known as component testing. Unit testing involves testing the individual components of a system to ensure that they meet the functional requirements and have the correct structure. For example, I conducted unit testing on the Appointment and AppointmentService classes, which are separate components. To perform unit testing, I employed three techniques: functional testing, white-box testing, and regression testing.

Functional testing focuses on testing the functionality of a component based on its specific requirements. During functional testing, the system is tested against various inputs to verify its behavior and outputs. The test cases are designed to cover different functional requirements, features, and user interactions. The tests are executed to validate that the software performs the intended operations correctly, produces the expected results, and handles errors or exceptions appropriately. In the case of the Appointment and AppointmentService classes, one requirement for the AppointmentService class was the ability to delete an appointment object. To test this functionality, I developed a JUnit test that created an appointment object. Once created, it would be deleted and then the test would confirm that the object had been successfully removed. By conducting functional tests on each component in the three services, I ensured that the codebase functioned as expected and met the specified requirements.

White-box testing, on the other hand, involves testing the structure and design of the software to ensure that the non-user-interactable parts of the code function correctly. Throughout the services I developed, I employed white-box testing techniques by creating JUnit tests to validate the input and output streams, the various method calls, and correct exception handling. For example, I developed a JUnit test for the ContactService class to validate the update contact method, which would create a new contact object. To do this, the method would call four setter methods in the Contact class, which is functionality that a user would not see. This test ensured that the method correctly called the setter methods within the Contact class to update a contact object. White-box testing techniques like this were applied to all the components developed in this project.

Regression testing is the act of testing components to validate changes or updates to the unit have not introduced new defects or caused unintended side effects. It helps ensure that the software maintains its desired behavior and does not experience a degradation in quality as it evolves over time. When developing the services in the beginning, I was utilizing an ArrayList to house the objects that the system was creating. It was identified that a hash map would be a better data management approach, so the services were updated to accommodate this. To ensure that this update did not cause unintended defects, testing was completed again on the services. regression testing is an essential practice in software development as it ensures the stability, reliability, and continued functionality of the software even in the face of ongoing changes and updates.

However, it is worth noting that there were other testing levels that were not utilized, such as integration testing, system testing, and acceptance testing. Additionally, specific techniques like black-box testing and non-functional testing were not employed. These testing levels and techniques typically occur in the later stages of the software development life cycle (SDLC) when component parts are integrated to form a system that needs to be accepted or when components are now interacting with each other. Each testing level has specific objectives that are achieved through various testing types, including functional, non-functional, structural, and regression testing. All these levels play a crucial role in ensuring that the final system functions correctly and meets the established requirements.

The structural testing technique, Black-box testing, is a software testing technique that focuses on evaluating the functionality of a system without considering its internal structure or implementation details. In black-box testing, the tester treats the system as a "black box" and only interacts with its inputs and outputs, without any knowledge of the underlying code or logic. Black-box testing techniques would be implemented once the system is integrated or when user interaction is implemented. These techniques are particularly useful in providing an independent assessment of the software, simulating user interactions, validating requirements, assessing user experience, and identifying potential security vulnerabilities. Executing this testing technique with white-box testing can provide comprehensive coverage and help ensure the overall quality of the system being tested.

Non-functional testing, in contrast to functional testing, concentrates on evaluating the characteristics of a system rather than its specific functionalities. Unlike functional testing, which verifies if the software meets specific requirements, non-functional testing assesses aspects such as performance, reliability, usability, security, and maintainability. This type of testing is important in system development because even if the system functions as intended and meets the requirements, it may not perform well under heavy load, lack scalability, or have adequate security measures. By conducting non-functional testing, software development teams can identify and address potential issues early in the development lifecycle, enhance the overall quality of the software, and deliver a reliable and efficient system to end-users.

When working on this project, I had to step out of my own way and approach it from the perspective of a tester rather than that of just a developer. This mindset allowed me to think through the issues that could arise from the code base that I was developing. By utilizing caution in my thought process, I was able to identify and plan through how classes and their services would interact with each other and ensure that the requirements would be met through any interaction within the system. For example, the Task class is utilized to create task objects which hold the task information. The Task Service class utilizes the Task class to create those objects, and I needed to verify that when a new or existing object is created or manipulated, I have thought through the various situations that could happen and address them. Ensuring that variables are unable to be utilized when null or if charater counts are too small or large, the system would handle the exceptions with grace and address the issue.

A problem that can come to light when testing code that you yourself have developed is a bias in its functionality and accuracy. A developer can grow blinders when it comes to the code they have developed, limiting their perspectives on possible issues or being defensive in its accuracy. By stepping back and approaching the project from the perspective of “how can I break it”, you are able to get out of your own way and develop test cases that ensure a successful and efficient system. An example within this project is the inclusion of input validation in the setter methods for the three object classes. As a developer, input validation could be part of the methods that call those setter methods, so they may not think to include input validation within them. If approaching the project from the viewpoint of how the system could fail, you would see that including input validation at all points ensures that a non-valid input will not make it into the objects from unforeseen situations. It is imperative that bias is limited when testing software at all times.

Being disciplined in the commitment to quality should be the basis for all software engineering professionals. Cutting corners when it comes to writing or testing code can have severe consequences for the software's performance, reliability, security, maintainability, and overall success. Not only can it hurt the software being developed, but it can also have ramifications for customers, users, and the company as a whole. Technical debt refers to the metaphorical concept in software development that describes the long-term consequences and costs of taking shortcuts or making trade-offs during the development process. Sometimes it can be unavoidable to develop technical debt when working on a project. This in itself is not an issue. The issue arises when that technical debt isn’t “paid off“. As an example, a project may have a timeline that does not allow for detailed testing to be developed for the entire unit. If that unit is then completed but not refactored after the fact to ensure there were any unseen issues, it becomes bad technical debt, or what I would call “unpaid debt”. Addressing technical debt requires a proactive approach and a commitment to ongoing code quality and maintenance. By acknowledging the trade-offs made during development, allocating time for refactoring, and prioritizing debt reduction, I can minimize the long-term negative impacts of technical debt on software projects.

**References**

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

Schmitt, J. (April 14, 2023) *Functional vs non-functional software testing* CircleCi Blog.

<https://circleci.com/blog/functional-vs-non-functional-testing/?utm_source=google&utm_medium=sem&utm_campaign=sem-google-dg--uscan-en-dsa-tROAS-auth-brand&utm_term=g_-_c__dsa_&utm_content=&gclid=Cj0KCQjw7PCjBhDwARIsANo7CgkiWvI2lsaKTb78_mZ--9bx8qoYNz6O3VPLlJclwKWx7ETZL25E0jQaAnujEALw_wcB>

Velimirovic, A. (March 3,2022) *Technical Debt: Definition, Examples, and Types* phoenixNAP.

<https://phoenixnap.com/blog/technical-debt>

*Difference between White Box testing and Black Box testing* javatpoint.

<https://www.javatpoint.com/white-box-testing-vs-black-box-testing>

Pittet, S. (na) *The different types of software testing* Atlassian.

<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>