**Module Seven: Project Two**

Stanislav Kovalchuk

Southern New Hampshire University

CS-320 Software Test Automation& QA

Ramsey Kraya

February 20, 2024

In my recent project to develop services for a client's mobile application, I used an individual approach to ensure that the final product perfectly meets the specified requirements. Each service I developed was accompanied by a set of detailed requirements describing operations and operational boundaries. These requirements have served as the cornerstone of my development efforts, helping to streamline the workflow aimed solely at meeting the needs of the application. For example, if we consider the “Contact” service, which was the first service that we developed. This service contained five different requirements for the “Contacts” class and three requirements for the ”Contact Service" service. These specifications required the creation of a “Contact" object with a mandatory contact ID string limited to 10 characters and to be immutable. Also, for the contact name, the limit was 10 characters and the name should not be null. In addition, a ten-digit phone number string was required, which could not be null, and an address field limited to thirty characters, which could not be null. After reviewing these requirements, I initiated the development process by compiling pseudocode as a guide. For example, for the initial requirement of an immutable and non-zero contact name, pseudocode can create an object of the “Contact” class and assign various values to it for testing, for example:

@Test

@DisplayName("Test Contact First Name")

void TestContactFirstName() {

Contact contact = new Contact(contactId, firstNameTest);

//less than 10 char test

assertAll("Contact First Name",

()-> assertEquals(firstNameTest, contact.getFirstName()));

//10 char test, null and more than 10 char test

contact.updateFirstName(tenCharFirstNameTest);

assertAll("Contact First Name",

()-> assertEquals(tenCharFirstNameTest, contact.getFirstName()),

()-> assertThrows(IllegalArgumentException.class, () -> contact.updateFirstName(null)),

()-> assertThrows(IllegalArgumentException.class, () -> contact.updateFirstName(longFirstName)));

}

Throughout the entire testing phase, I systematically checked the compliance of the developed functionality with the specified requirements. This included creating test objects, attempting operations with invalid values or actions, and verifying that the system reacted in the expected way. By accurately reflecting the requirements in both pseudocode and test cases, I have ensured comprehensive coverage and reliability in the final implementation of each service.

In an effort to ensure the effectiveness of JUnit tests based on the percentage of coverage, I experimented with various methods. Initially, I tried to create a single class to run all the tests simultaneously, intending to run this class for coverage analysis. However, due to my limited experience with this approach, I have encountered difficulties. However, I quickly adapted using an alternative method — I just right-clicked on the package containing my code and executed it to analyze the coverage. This method gave satisfactory coverage results, allowing you to get an idea of the degree of code usage during testing. To preserve the integrity of the coverage results and prevent distortions caused by the inclusion of test code, I applied a simple approach. Realizing that the percentage of coverage covers both production and test code, I subtracted the portion related to the test code. This allowed for a more accurate assessment, showing that more than eighty percent of my class's code has been dynamically tested.

In an effort to achieve technical soundness and efficiency, I have used several strategies. For example, I strategically used print functions to track object instantiation and function execution order, helping with debugging and understanding the code flow. After the service was ready and everything worked correctly and without errors, I deleted all print functions so as not to clog up the service code with unnecessary code.

This project used a wide range of software testing methods, including both static and dynamic methods to ensure thorough verification of the developed services. Static testing, which included analyzing the code without executing it, played a key role. For example, by declaring the “atomic long” used to generate the identifier, it was guaranteed that subsequent attempts to change the identifier would be eliminated, thereby meeting the requirements. The main focus of these tests was to verify that the input data is within the established boundaries and that the structure of each component exactly meets the requirements set out in the specification document.

Another point of view classifies the methods used as a "black box" or "white box". Black box methods, also known as specification-based methods, were used to verify the external behavior of the system. Conversely, white box methods, or structure-based methods, delved into the internal logic and structure of the code to ensure correctness. Structural testing, a form of white-box testing, has been integrated into the project to determine the extent to which testing coverage has been achieved. This included creating a coverage report to assess how thoroughly different parts of the code were used during testing, which gives an idea of potential gaps in testing coverage.

However, some software testing methods, such as non-functional testing, were not used in this project. Non-functional testing, which includes evaluating aspects such as performance, reliability and usability, has not been conducted. For example, comparing the system under test with certain industry standards to assess its non-functional characteristics was not part of the testing strategy used in this project.

Using a combination of static and dynamic testing methods, along with the "black box" and "white box" approaches, the project was aimed at providing comprehensive verification of the developed services, thereby improving the overall quality and reliability of the software.

Throughout this project, my thinking has been characterized by a cautious approach, primarily due to my status as a beginner in JUnit testing. At times, when I faced problems such as trying to implement a class to run all the tests, my understanding of the scope and related concepts was not fully developed. Consequently, I resorted to elementary methods, such as print the contents of an object, to test the functionality of the code, which reflects my pragmatic approach to solving problems.

In the absence of external code analysis, it could be tempting to bypass testing of certain aspects, which could potentially lead to the identification of defects. Recognizing the importance of discipline in implementing an effective and thorough development process, I have prioritized comprehensive testing, realizing that shortcuts can compromise the quality of the final product.

Even though I was confident in my understanding of concepts such as the consequences of the final assignment of a variable, I realized the importance of carefully testing the code. These decisions, based on a careful approach rather than haste, will influence my future decisions when developing the project.

Moving forward, I intend to strictly adhere to the specifications in future projects, recognizing that creating a final product that meets the requirements requires a disciplined approach. A method that I particularly approve of and which I will implement in future developments is to use the requirements as a guide to create pseudocode that promotes a structured and systematic approach to coding.