**Nicholas Rusinski**

**Robert Tuft**

**CS-320-11162**

**Jun-21-2024**

**Summary**

As a developer for Grand Strand Systems, I was recently tasked with developing services for a mobile application. This report outlines my approach to unit testing these features, my experience in writing these tests, and reflects on the testing techniques used throughout this project. The objective of this report is to provide a comprehensive overview of the testing methodologies used and their alignment with the software requirements.

My development task was centered around three main features that will add functionality to the application. The first feature was a contact class that allows a user to store contacts by name, phone number, and address. Next was a Task Class that enabled a user to store daily tasks by name and description. Lastly, an Appointment Task helped users track their upcoming appointments, stored by date and description. Implementing these features was essential to the overall functionality of the application. To ensure their proper functionality, a unit testing library called JUnit 5 was deployed to automate the specific requirements of each feature. This works by using syntax from the JUnit 5 framework to build test cases that run code and verify that the code behaved as intended.

The general approach I used to apply unit testing to these features was first to understand the JUnit 5 design, such as assertions and different types that could be used. This provided a solid foundation on how to apply unit testing efficiently. Next, I focused on the specific requirements expected to ensure the smooth operation of the application. This highlighted critical areas and provided a framework for identifying test scenarios. The focus then shifted to creating individual test cases to cover various requirements of the code. This process was repeated, focusing on the indirect needs of the application that underlie the direct requirements. The last step before executing the test was to establish the setup environment where objects and data that apply to all test cases are initialized. Finally, executing the test cases and correcting any erroneous behaviors flagged by the tests.

The functionality of the different features and classes called for unique approaches within the generalized approach outlined above. Starting with the Contacts feature, my approach was to first test the creation of a contact object using assertions to check the success of each variable. This was critical for subsequent test cases, where each requirement was tested by creating additional objects with incorrectly populated variables that would throw exceptions if the object was created with invalid data. Each requirement of the class was tested in this manner, ensuring that the object was created in a way that aligned with the software requirements. Since the creation of the contact objects relied solely on the constructor, this method proved effective and guaranteed a high coverage percentage of the feature with minimal testing code.

The Task feature also required a different approach to align with the software requirements. To test the ability to properly add, delete, and update tasks effectively, I used setup annotations to run code before each test, initializing the objects needed for each test case. This ensured consistency in each test case by isolating each test from the others and starting each test on the same ground. It also improved the readability and maintainability of the test environment, ensuring that test cases could be effectively designed to meet software requirements.

Next, the Appointment feature of the application required a unique approach due to its interaction with date data types. To meet the requirements of these tests, individual assertions were used to test the behavior of the code. Assertions like assertEquals were used to check objects and variables after an update and/or creation. Due to the nature of date data types, assertions that checked whether an exception was thrown were used to test dates with hidden time components, making date comparison impossible for an equal’s assertion.

Finally, to demonstrate total coverage of the features, I used a testing class that employed a testing suite. This JUnit 5 annotation allows multiple test classes to run from a single hub. Testing all six-unit test classes at once allowed for easy visualization of the total coverage of the three features, ensuring the testing coverage requirement was successfully met.

Overall, I found that using the Junit 5 framework was simple and convenient, with an abundant amount of content and examples readily available online. It enabled tests to clearly and effectively verify that the code was technically sound. An example of this is in the Appointment feature, where a test case was set up to check if it was possible to delete a non-existing appointment. The test name indicated its intended use, and with a simple assertion checking for an exception, it confirmed that the code handled the invalid request in a way that did not crash the program. The simplicity of this line demonstrates the effectiveness and efficiency of the JUnit 5 testing library.

**Reflection**

Functional software testing was the focus of this project. Functional testing involves testing the core functionality required to meet customer specifications and requirements. This is demonstrated by testing only the features of the project, not aspects like compatibility or hardware requirements. Another testing technique used was unit testing. Unit tests are used to test the individual components of the application with automated scripts. Unit tests are part of a broader testing technique called white-box testing, where the tests are closely integrated with the source code, requiring testers to understand the project at a developer level. These testing techniques are practical, particularly during the development stage of an application, because they require testers to be involved and understand the source code. They can easily be automated, allowing for rapid and continuous testing, and they do not require code execution or a complete working application. These aspects make them ideal for this project, which has been designed but not fully developed and will allow the team to continuously verify that these features work when integrated into the developed application.

The mindset I used to ensure the success of this project was to meticulously understand the code requirements. This was straightforward for this project due to its white-box nature, allowing me to design the code and corresponding tests. Caution was also exercised in selecting the appropriate unit test assertion for each case. For instance, an assertion checking for null ensured that required variables were populated, confirming the successful execution of the test by aligning directly with the testing requirements. Limiting bias was another important aspect of my approach. As the code creator, it was crucial to include additional testing and not assume that passing tests meant everything behaved as intended. An example of this is the delete function used in each feature, where I tested different scenarios to cover multiple deletion perspectives.

In conclusion, maintaining a disciplined approach to testing and code quality is essential for delivering reliable software. By exercising caution, reducing bias, and prioritizing quality, I ensured that the mobile application met high standards of functionality, reliability, and user satisfaction.