**Module 7: Summary and Reflection**

Christopher Richards

[Christopher.richards4@snhu.edu](mailto:Christopher.richards4@snhu.edu)

04/16/2023

1. **Summary**

For the project submitted last week, Grand Strand Systems tasked our team with creating a mobile app that allows for the creation and scheduling of appointments. The application consisted primarily of three backend modules: A Contact class, a Task class, and an Appointment class. Various requirements for each class were given to the team to be incorporated in the design of the classes. In addition to designing and implementing these three classes for Grand Strand Systems, we were tasked with designing JUnit tests to perform dynamic testing of our Contact, Task, and Appointment classes.

When designing these tests, careful consideration was taken to ensure that the required criteria for each class were included in the JUnit tests that were to be performed. At the very minimum, these requirements had to be included within the coverage of the tests. If a JUnit test had 80% or more coverage but missed a requirement of the project, the JUnit test could be considered a failure.

The requirements for the Contact class were the following: The Contact object should have a unique ID, no longer than 10 characters and may not be null or updateable; the Contact object should have a first and last name that can be no longer than 10 characters each, and cannot be null; the Contact object should have a phone number that is exactly 10 digits and shall not be null; and finally the Contact object shall have an address that is no longer than 30 characters and shall not be null. In each of these sets of requirements, it can be noted that none of these fields can be null. One JUnit test may assert a throwable if the actual value of the field is a literal null. But there is also a practical case for a “null” value as well. Should the application throw an InvalidArgumentException if the field is, say, an empty string? Certainly, Grand Strand Systems doesn’t intend for this kind of behavior. If the value of any of these fields for the Contact class is an empty string, it isn’t null because it is a string. We can effectively treat this edge case like a null value. Thus, the design of the class and the JUnit test should assert and test for the assertion of an InvalidArguemntException. The properties for the Task and Appointment class follow the same suit.

When it comes to the ContactService, TaskService, and AppointmentService classes, the JUnit tests need to have a slightly different design. These classes do not have the same constraints regarding fields for the class. Take the ContactService class as an example. The requirements are: the ContactService class shall add Contacts with a unique ID; the ContactService shall be able to delete contacts per contactId; and the ContactService shall update contact fields per contactId. The ContactService does not contain a direct reference to any contactId it contains. As such, The design of the class and JUnit test must assert null and test for the assertion of null when adding a Contact to the ContactService, by first attempting to retrieve a Contact with the same Id as the Contact being added. Similarly, when trying to delete a Contact from the ContactService, there must be a Contact to delete. In contrast to the test for adding a Contact, we must assertnotnull when retrieving a Contact to confirm the contact exists before deleting it. The other service classes follow suit in this pattern.

As stated previously, the goal is to have all defined requirements covered by the JUnit tests. Additionally, we were given parameters to have 80% coverage of all classes under test. Without some careful forethought, this task can be more difficult than originally intended. When running the initial JUnit tests without any planning within the JUnit test setup under the project, the JUnit classes will also be considered when calculating coverage. As such, it may be common to get test coverage results of 70% or less. The goal of the JUnit testing is not to test the test class itself, but to test the application class that will be used. Thus, it becomes important to exclude the JUnit classes from within the test coverage configuration before running any test configurations. One other issue that may affect test coverage reporting is how the tests are designed. It may make sense to run a test of the Contact class’s requirements when adding a Contact to the ContactService. It stands to reason that this is a thorough method of testing. We want to make sure that the requirements of the Contact class are adhered to when adding a Contact to the ContactService. But this can cause the JUnit test to underreport the test coverage, as performing a test against the requirements of the Contact when adding to the ContactService effectively prevents the unit test from running for the Contact class. This will cause all branches of logic for the Contact class to report as not being covered. Once I figured this out, I was able to achieve 100% test coverage for the project. It is important to remember that the overall coverage isn’t the only metric to determine adequate testing. If your class does not have the design in place for the class requirements, it won’t report as not being covered, rendering an artificially high test coverage and leaving a possibility of defects within the class.

1. **Reflection**

The software testing techniques that were employed within the project were dynamic testing techniques. Dynamic testing is a testing technique in which application code is tested at run-time, or during execution of code. Dynamic testing is designed to test the behavior of the application (Hambling, Morgan and Samaroo, 2019). The JUnit tests that we designed and implemented run the class code and tests various inputs and outputs of the class. The dynamic testing implemented using JUnit tests is a type of dynamic testing called unit testing. According to Hambling, Morgan and Samaroo (2019), unit testing occurs during development to verify requirements and functionality and is primarily performed by developers and testing analysts. Other phases of dynamic testing include integration testing, system, and user acceptance testing. Integration testing tests the functionality and interaction of the codebase as it is integrated with the larger codebase (Hambling, Morgan and Samaroo, 2019). System testing tests the application’s interaction with other applications and technology stacks as part of a system. And finally, user acceptance testing is dynamic testing in which the application is tested for completion and requirements coverage.

Another testing paradigm is static testing. Static testing is testing in which the item to be tested is examined and scrutinized without executing any code (Hambling, Morgan and Samaroo, 2019). With static testing, all aspects of the development process are susceptible to static test, including business requirements documents, application design and even testing design.

When working on this project for Grand Strand Systems, it is important to maintain a proper mindset when designing JUnit test cases. The goal of unit testing is not to affirm the absence of defects, rather the goal of unit testing is to identify any outstanding defects. It can be easy to write unit tests that test basic functionality of your application and forget to test the interactions of different components of the application, such as the interaction between the Contact class and the ContactService class. We can test the requirements of the Contact class and test the requirements of the ContactService class, but what happens if we write code that satisfies the ContactService class requirements but violates the Contact class requirements. These are important to test as well and could very easily be forgotten about when designing unit tests.

When designing unit tests, it’s important to try and keep a bias toward the application code out of the test design. As the software engineer who designed the application, you will have a certain understanding and expectation for how the application should be interacted with. This bias toward an expectation of the software could easily affect the design of unit testing. To help eliminate this bias, it may be suitable to have a different software engineer or test analyst design and implement unit testing in your stead. As someone with no predetermined bias toward the software, they are more capable of writing unit tests that may find edge cases in which defects may arise.

As a software engineer, it is incredibly important to maintain a certain level of professional discipline when designing, implementing, and especially testing software. Undiscovered defects can lead to monetary consequences in the lifespan of a software. According to Hambling, Morgan and Samaroo (2019), the cost to remedy a defect once it has made its way to a production environment is significantly more costly than if the defect is detected and remedied early within the development process. Undetected defects don’t just come at a monetary cost. Defects that find their way into a production environment may also come at a cost of reputation and customer confidence.

In summary, it becomes vital to take a nonbiased, thorough approach to test design and implementation. The use of many different types of testing techniques and strategies early and often within the software development lifecycle gives the greatest chance of identifying software defects early. Identifying defects early and implementing remediation before the production stage of the software will save time, money, and reputation. This will give customers the confidence needed to continue a fruitful partnership with Grand Strand Systems going forward.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software Testing : An*

*istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.