### Southern New Hampshire University

### Summary and Reflections Report

Zackeriah Ibarra

### Neil Kalinowski

CS-320-T2646 Software Test Automation& QA 21EW2

12/11/2021

**Summary**

By in large, the testing required, and used, for each of the given are similar. Each one requires that variables have specific values, which requires that value input checks that the input is the correct length and not null. This is completed in is done in several phases. The first is verification of the basic functions of the code work; getters, setters, and constructs. This is done in the base class of the given program. For example, AppointmentTest uses set values that should pass to construct objects and search them to verify the construct built correctly and that the value is set as expected (Ibarra, 2021). Next is to ensure that the program does not allow for values outside the requirements to be set. This is completed by testing known values that should fail based on the requirements. Throughout each JUnit, there are sets of Fail Tests and Null Tests that attempt to set invalid values to each variable and pass when the correct exception is thrown. After valid object creation is proven, the final step is to verify the deletion and multiplicity of unique objects. Deletion is verified by creating an object, deleting it, and then searching for the same object by its unique identifier. If an exception is returned that it could not be found, the test pass. Multiplicity is tested by creating multiple objects and confirming that each is unique from the others. Overall, the testing done covers all requirements set for the program, and the fact the test coverage is 100% states that all these tests pass correctly. While there be more intensive and comprehensive testing that could be added, the testing set covers all requirements.

Writing this type of code was a struggle for me at first. Understanding how the testing code works and then deciding how to implement it to confirm the requirements for the program were met took quite of bit of trial and error. The first issue I had ultimately came down to having an excessive number of test values in my code. I had used specific strings to test variables in each of the test functions. This led to very messy code and made it difficult to identify issues in my code. I resolved this by making a set of defined strings from the start that would be used in each case, this made it much easier to identify issues as all the variables had the same name, and easier to confirm that the variables aligned with the requirements they were being used to test. Once I made these changes, I realized that each test case was similar in structure. Essentially each would assign a test value to a variable and ensure that it was correct. From this point, I used the same functions, with a slight variation to test separate functions. Going forward from here, I realized that negative testing was required to ensure that incorrect values also failed. I applied the same principle, I built one function that correctly verified that a given incorrect variable fails, then modify the function to test the other fail cases. At this point writing the tests for deletion and multiplicity was straightforward.

**Reflection**

Of the many testing techniques considered when writing the tests, I only used a few. The requirements were direct and did not require complex testing to ensure that the code performed the tasks expected. I primarily used the Black Box testing technique, boundary values. I used specific test values, known to be in and outside the boundary of the requirements, and designed test cases to pass when the correct outcome is evident. While only using one type of testing may leave the potential for gaps in the coverage, however after considering the requirements and the other techniques, boundary values proved that the code met the requirements set. Techniques such as status transition, decision tables, and use cases do not apply very well to this code. The program or any of the pieces does not change states at any point, so there isn’t a need to test functions under the conditions of a status change. Each input only impacts a single outcome of the program, and there are no rules or cases that imply that a particular set of inputs should have a specific effect, so decision tables are unnecessary. Finally, there is no requirement to restrict different functions to types of users, so user case methods would not be practical. Equivalence partition would potentially be usable testing in this code. One of the main requirements throughout all the programs is the size of the string each variable can be set to; I could have set up testing parameters that cycled through various lengths and combinations of characters in the string and failed and stopped once the combinations reached outside the limits of the requirement. While this would have been much more thorough testing, the tests that were completed give nearly the same assurance and are less time-consuming, both writing and during the test run.

My mindset while writing this code was to attempt to complete a couple of things. I wanted to ensure I capture all the test cases. To make sure that I understood the mechanics of the program code to ensure that the test cases associated were properly structured. All this requires the person to understand what each part of the code is doing. Like areas where each unit of the class has some specific requirement to be able to function. Bias is a major factor that can cause any endeavor to fail and should be considered when not only writing test code but in all aspects of SDLC. Like when creating the variables in a class, you might have the confidence since you were the one who created it, that the code is effective. We all should try to limit that when it comes to software testing. It is very important not to cut corners when it comes to writing or testing code as millions of money losses can be avoided and even loss of life can be avoided by testing the codes properly and ensuring everything is working perfectly before shipping the code to production.

**References**

Ibarra, Z. T. (2021). Project One. program documentation, Cortez; Personal Work.