**Summary and Reflection Report**

Jerry Davidson

Southern New Hampshire University

CS-320-T5449

Prof. Bryant Moscon

June 19, 2021

## Alignment to Requirements

## I used test-driven development to create the contact, task, and appointment features. The first step was converting the requirements to tests. I read through each requirement and aligned the tests to its specifications. For example, each feature required an ID, so I ensured that the tests covered that the ID was not null and met the required length (could not be greater than ten characters). So, for the subclass that would later be used by the object class (i.e., task is a subclass used by the object class task service), I ran tests until each of the requirements passed.

Creating tests for the object class was the next step. My process behind creating test classes developed with each application. I initially found myself coding the business logic inside of the tests, which was not the correct approach. Then an “aha!” moment came. The advantage of test-driven development presented itself—method names could be created before the actual code was written. Below is the object class code I used for the appointment feature.

Graphical user interface, text, application, email

Description automatically generated

Being able to predefine the methods eased the coding process by outlining the object class requirements. From the appointment object class code above, the methods for deleting and adding were created before the actual code. The desired output was defined as well. It was important to leave out business logic in the tests. Tests were run until the method call executed and gave the test back the required output.

## Effective Tests

## Coverage was the final step in my development. It ensured that each test covered the methods within the code. With test-driven development being used as an outline to the methods, testing for coverage took little effort. Issues only arose with how many cases a method covered. This occurred when I used the less than, greater than and syntax for equality (“==”). To ensure the cases were covered, I used the built-in Java methods to tell if a value was less than or greater than, and to check for an equality. This is also displayed in the object class image for appointment. The other features followed the same setup.

## Technically Sound Code

The important part for making the code technically sound was using easily understood function names and variables. I decided on the names and how they should function when I created tests. For example, the task function, “getID” returned the ID for a task and the variable name was used for the task name (a screenshot is attached below). For brevity, I created the helper function “checkStringGreater”, to check if initialized constructor values were null or did not meet length requirements. Graphical user interface, text, application

Description automatically generated

## Strategies for Efficient Code

Testing for how well my code works was a bit tricky. I initially thought I only had to get tests to passed. I decided to run coverage and found out that I did not cover all cases for my functions. Specifically, the task service test function “testRemove” checked that a task was not removed by checking equality with the returned integer (check the screenshot below). Note that the variable unfound is equal to -1 and the “findTask” function returns -1 if an object ID is not found.

Graphical user interface, text, application

Description automatically generated

This only provided two of four test cases, so I decided to refactor to the code below.

A picture containing text

Description automatically generated

The new function for finding tasks returns null, allowing all cases to be covered.

## Techniques Employed

For each milestone, I used the same testing techniques. I started off with creating tests before writing the application code, known as test-driven development. I ran the tests as I coded, and I stopped once all tests passed. One important part about this was ensuring the tests were not becoming a part of the application code. Also, each method from the requirements was tested, which is unit testing. It would be tough to identify errors if the code was checked.

Regression testing was also important for changes made within the source code. There were times when I had to go back and adjust code due to inadequate coverage. For example, using less than, greater than or equal too left out test cases, leading to poor coverage. To determine if an item was added to a list (each milestone involved a list to add a task, appointment, or contact), I used the array method to check if the list was empty and asserted this to be false.

## Other Techniques

Black-box testing tests the functionality of a system (García, 2017, p. 22). None of the milestone had a frontend (the part that a user interacts with), so GUI testing was not used. One method that should have been included is random testing. This tests the system with random and bad data for failures. For the appointment service, this would have been great to ensure that the date input was accurate. Dates had to be in the future, and it can become very confusing when considering time zones.

The data structures I used for the services were very basic as well (array lists). I went with the simplest structures because there was not a heavy data load. This made performance testing unnecessary since speed would not impair the services. Uses and Implications of Techniques

Test-driven development is best for situations where the requirements of the application are known. This helped tremendously with deciding what methods would be used in creating the services. Regression testing was also important since each service stored a list of objects (task, appointment, or contact). For applications with heavy integration, this eliminates a lot of time doing manual checks for functionality. If the application had a user facing side, GUI tests would also save time in ensuring proper functionality.

## Bias

Completely eliminating bias was impossible since I was the only person behind the work. From my experience with programming, I find that old work looks new to me. With this, I would complete the work and then check it later in the week. This allowed me to relearn what I coded and identify any issues I overlooked. Other than that, strictly following requirements by writing tests ahead of time eliminated some bias. As a software developer, testing and writing my own code could lead to lots of bias. When I first wrote the tests, I coded before writing the tests. Then when I wrote tests, I added methods to help the tests pass. This was an inefficient approach. I soon decided to write tests before coding. This way I had to code until the tests passed—without changing the tests if they correctly covered the requirements and provided efficient coverage.

## Discipline

## Cutting corners during the programming phase can be costly! Writing proper unit tests ensures that all areas are and can be tested. This is very important for regression testing. As an application evolves bugs become harder to identify without proper tests. Testing increases productivity because developers do not have to manually check each part code. To eliminate technical debt, I should create clean and reusable code. This means if the same logic is used multiple times, it should be extracted as a method. If the logic is becoming unclear, it will be hard to maintain. If the code cannot be simplified, comments should be used to explain it. This helps future developers maintain the code and reduces the amount of review I must do before updating the code if a bug is found.

**References**

García, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.