**CS320 Project Two**

Charles M. Haines II

Department of Computer Science

CS-320 Software Testing, Automation, and Quality Assurance

Dr. Albanie Bolton

October 14, 2022

I used the software requirements as a starting point when I was designing the various features of the application that I developed. Having the list of software requirements is what allowed me to design and ultimately test the software by defining what the software should do. While I did not write my test cases prior to designing the code for the application, I wrote the code with the requirements in mind since this would ensure that the test cases would pass once the were written and executed. For example, the id field for the various objects I created was supposed to be unique and immutable. To ensure that the id could not change, I created the id as a private field which could only be set by the object’s constructor. By ensuring that only the constructor could set the id value, this means that the only way to change an object’s id is to replace it with an entirely new object. To ensure that all the requirements were met, I also ensured that when someone calls the code to construct a new object, that any attempts to create an object with parameters that did not meet the requirements would result in an error and preventing the object from being created. The appointment description was required to be fifty characters or less. If you attempt to create an appointment with a description longer than fifty characters, an error occurs that prevents the creation of the appointment.

I know that the quality of my Junit tests was good because I ensured that all the software requirements were met. One way to measure a test is called coverage. This is a percentage that shows how much of the code in question is being tested. While I did loose a few points due to coverage being less than 80 percent, this is mostly because the tests were not run concurrently. When instructing Eclipse to run all tests in the tests package, the tests cover 100% of the code for the objects being tested. The only code that is not covered when running the tests are the tests themselves. Since I ensured that the tests cover all of the requirements to make sure that valid objects can be created and work as expected as well as that attempts to create invalid objects result is an error message being displayed combined with the fact that the test cases cover one hundred percent of the code for the various objects, the test cases were effective.

I ensured that my code was technically sound by testing the objects such as Task and TaskService, to verify that they function as expected. The first test in the TaskTest.java file, which starts on line 24 called testTaskObjectSanity was designed to simple create a simple Task object with valid parameters and then test each of the fields to ensure their values were set as expected. This may not sound like an important test, but it is the first test in the suite because knowing that the object works as expected is paramount and should be confirmed prior to testing any other aspects of the object code. Similarly, the TaskServiceTest class contains a test on line 41 titled testCanRetrieveTaskObjectFromService which again simply tests that the Task objects that are contained within the task service can be retrieved successfully and that their data contains the values that are expected. I used a setup method in this test class which adds three example Task objects to the TaskService list to perform the various tests. By confirming that the example objects were created successfully, can be retrieved, and that their example data matches what is expected, I know that the code for the TaskService class is technically sound prior to testing any additional functionality.

I ensured that my code was efficient by finding ways to make my code more efficient when possible. In a previous assignment, in each of the methods I created to update an object, I wrote out the code to cycle through the list and find the object matching the supplied id in each of the methods used to update the various fields. I realized that this was inefficient and opted to try to do this more efficiently. On line 39 of TaskService.java, you can see that I created a method to get an object by supplying an id string. This method returns a Task object if the object is found in the list and throws an exception if the object is not found. You can see that in the update field methods such as updateTaskDescription, I make use of this method to find the object with the specified ID. Since the getTask method already throws an error when a matching ID was not found, I did not need to spend any time writing similar code in the updateTaskDescription method. The updateTaskDescription method can be located on line 78 of TaskService.java. I was able to confirm that this worked as expected with my test cases. The test case called testCantUpdateTaskDescriptionWithInvalidId on line 148 of TaskServiceTest.java tests that supplying an invalid id when attempting to update a task’s description results in an exception being thrown as expected.

One of the primary techniques I used when constructing unit tests is called equivalence partitioning. Equivalence partitioning consists of dividing the types of input into distinct categories or partitions. In my unit tests, I decided it would be easiest to create partitions for both valid and invalid types of input. After establishing the kinds of input that would be considered either invalid or valid, I was able to construct test cases that tested the various types of input from each partition to ensure that the correct output or response was given. For the appointment object I created in the module five milestone assignment, a description is considered valid if it is neither too long nor blank (null). I constructed tests that used a valid description, a description that was too long, and a description that was blank to confirm that only a valid description would be accepted when creating a new appointment object. Creating these partitions allowed me to test the functionality of the function being developed while writing the minimal number of test cases needed to confirm the program functioned as expected in each circumstance.

Another technique that software testers use is called boundary value analysis. While I did not make use of this technique in my milestone assignments, boundary testing is typically used when testing functions that have some type of valid and invalid numerical ranges that can be tested. For example, if a program’s function should output a number in a certain range, boundary testing can be used to write test cases that verify that the output of the function is within the expected range. Boundary testing is performed by writing test cases that test near the boundary values that differentiate between valid and invalid input. As an example, if I am writing a function that calculates a percentage which is stored as a decimal, the expected range of numbers should be between zero and one hundred. To test this using boundary value analysis, I would perform tests near the values of zero and one hundred to test that the output of the function does not cross these boundaries. In this example, I might check to make sure that the function does not return a value of 2 or -1 since these numbers are outside of the expected boundaries of the function. Testing values that are on both sides of the boundaries helps minimize the number of test cases by focusing on the areas that are most likely to cause issues.

The practical uses of these techniques are what allows software testers to both verify the functionality of software that is being developed as well as finding issues that the developers may not have noticed when writing the code. Using techniques such as boundary value analysis and equivalence partitioning, software testers can make sure that the code written by developers meets the requirements of the project and avoid writing unnecessary duplicate test cases. Since there are many types of software that accomplish different tasks, different techniques can be used to write the best possible test cases for that software project. If a software function being developed does not deal with numeric input or output, equivalence partitioning may be a better choice over boundary value analysis since boundary analysis is typically associated with testing numerical values. While I have only talked about two commonly used software testing techniques, there are many techniques used for testing and an experience software tester will know which techniques will work best for a given software application.

I employed caution when both designing the objects for the application as well as when designing the test cases to ensure the objects basic functionality. By ensuring that my objects met the requirements at the time of their creation, it made the testing phase go very smoothly. I had to have an idea of what the objects are supposed to represent as well as how the objects would interact with other objects to design the objects like the Appointment object in a way that would meet the software requirements. The service objects functioned like containers that held instances of the objects and contained methods to add, delete, and modify the contents within the container. Since the service object had methods to change properties of the various objects within the container, I called the various setter methods on the objects which did a check to ensure that the new properties met the requirements prior to changing the properties of the object.

I tried to eliminate bias when reviewing the code by approaching the tests not as the developer of the application, but in the mindset of a software tester who has not worked with or seen the object’s code. I did not look back at the actual object code when writing the test cases unless the test failed. A test failure indicated that something in the object code is not working correctly. If all of tests were successful, there would be no need to dig into the code itself. Bias can be a factor to think about if a developer is tasked with testing their own code. It is easy to overlook something that you authored because you have an idea in your mind of how you assume something is going to function. When writing the tests for the appointment date field, I could not figure out why my tests were not working as I expected. After some troubleshooting, I quickly realized that the object code worked correctly, I was not putting the year in the format that Java was expecting in the test cases which caused the tests to behave in a way that did not make sense at first. Once I realized that the year value, I thought I was setting to 1900 was set to 3800. The date I was attempting to set in my test case was not recognized as being in the past because it was being set to a value in the future!

As a software engineering professional, it is important not to cut corners because software engineers are responsible for the performance of the software they deliver. Not every person takes the time in life to learn about things like programming languages and software development. When people use software, they expect that the developers know what they are doing and not doing things that are unethical. Just like you would trust your mechanic to properly fix your vehicle, you expect that software that is developed either commercially or as part of an open-source project is designed efficiently and tested to ensure that it functions as you would expect it to. I plan to avoid technical debt as software engineer by designing software using known and proven techniques and design patterns as well as using the software requirements as a guideline. One thing I did during this project is always having the requirements on one of my monitors so I could refer to them at any time when designing the various objects that comprised the application. By sticking to the requirements when designing the objects, the test cases were easy to write and worked the first time the tests were run. By getting the design correct initially, most of the tests passed the first time they were executed.