**Journal: Unit Testing Approach and Writing Junit Tests**

Justin Starr

Department of STEM

CS 320 – Software Test Automation & QA

Professor Ramsey Kraya

September 24, 2023

**Journal: Unit Testing and Writing Junit Tests**

For this module and the previous module, we have been asked to write java classes for an application. With those classes Junit tests were written to validate that all class files for the application work as intended. The following is a short analysis of the Junit tests that were written to validate the Contact, ContactService, Task, and TaskService classes.

The testing approach taken was very much aligned with the requirements for the assignments in that I had to ensure each class contained the required objects and their associated attributes. Each object’s attributes had specifications that when used or created, they must meet the specified requirements. When writing the tests, the tests were written to validate that when object’s were created, they met the specification set forth in the requirements. Additionally, in the service classes, is was required that methods be written to support adding and deleting contacts and tasks based off of a unique ID. If the ID was already being used then a new contact or task could not be created with the same ID. Also, Contacts first and last names, phone numbers and address could be updated based on the contacts unique ID. The same applied for tasks whereas tasks could be updated based on their unique id with a new task name or task description. All of the testing had to include tests that validated that these methods worked properly. For example, when a contact is added, it must be unique, it can not be null and it cannot be greater than 10 characters. The same applied to tasks. A task Id must be unique, cannot be null, and cannot be greater than 10 characters. Therefore, Tests were created to ensure that the application added contacts only if these conditions were met. The same applied to tasks and both the contacts and tasks associated attributes. Once all the requirements had been met, I was able to incorporate tests into the code such as throwing IllegalArgumentException when Contacts or Tasks were created that did not meet the specifications of their object’s attributes. Additionally, tests could then be written to see what would happen if objects were created outside of the attributes requirements (would they fail or would object’s still be created or allowed to be updated). While the requirements did not specify how the tests were to be written, they were absolutely a basis for what tests needed to be created to ensure the application worked properly.

The coverage percentage for the ContactService and TaskService classes shows the percentage of line coverage for each of the classes. It shows the percentage of each of the class's instructions that were covered by Junit tests. While Junit testing coverage does not prove that all test cases are covered, they do show that if you get 100% line coverage for each class all of the instructions are in some way, tested. However, this does not mean that it tests, for instance, that a contactId is not null or a taskId is not null. There are several more tests that need to be written to prove that the code works correctly outside of the scope that line coverage shows. Using line coverage can also be beneficial for determining unnecessary or unusable code.

The way that I was able to ensure that my code was technically sound was to write tests that showed it would do what it was supposed to do and also to write tests that would expect errors if the code did not do what is was supposed to do. The more tests that are written, the more you are able to prove that your code is technically sound. While it is not necessary to write a test for every possible outcome, we are able to look at what is expected from the requirements and test below the requirement, in between or even the exact requirement, as well as a test for what happens above or greater than the requirement. An example of this can be seen here in the TaskTest.java (lines 24 through 54). These five tests prove what taskId will be accepted and which taskId will not be accepted. Line 25 tests if the Id is null and passes because the passed in ID is null, proving a taskId cannot be null. The same applies as the next test creates a task with a valid taskId. The returned tasks ID is NotNull therefore the test passes again. We continue the testing for the taskId to ensure it is not too long, that it can be shorter than 10 characters, and that it can be exactly 10 characters. By applying these same types of tests to all of the object’s attributes for both tasks and contacts, we can ensure that the code is technically sound.

In the same way that code can be tested to validate that it is technically sound, code can also be tested to validate its efficiency. An example can be found by looking at the TaskTest.java lines 109-125 for the deleteTask method. We already know that all the task attributes have been checked and can only be used if they meet the object’s requirements. But can we use the objects the way they were intended to be used? These two tests for deleteTask show that they can be. One test checks to ensure the method actually deletes tasks, while the second test shows that if the taskId is not found, no task is deleted. The same can be applied to all methods for all classes by checking that they do what they are intended to do. This helps ensure that our code is not only technically sound but also that it is efficient. Lastly, we can increase efficiency by ensuring our code follows best practices in all aspects of our code.