Andrew McPherson  
CS-320  
4/17/2022

**7-2 Project Two**

The testing approach of all three features, these three being the Contact, Task, and Appointment classes, were all built to done nearly identically. The only difference between these three features and their classes is the limit of character length, variable names, and other minor things like the inclusion of testing the date for the Appointment class. Aside from these changes any approach of testing for one is nearly identical to the next. As for the alignment with software requirements, this was done by ensuring that within their construction the variables used in these objects were not capable of being created as null or beyond the character limit requirement for all different variables. I tested to make sure that the creation of the objects was correct and that each variable in the object was defined as it should be given the parameters passed to the constructor. But much of the testing for all three of the object classes was designed specifically to adhere to the requirements fully. This also carries over into the Service class for each three as well, where there were often requirements of being able to add, delete and update each of the contact, task, and appointments by the ID. Much of the testing comes from confirming the code for these classes is correct and does what it needs per the requirements, and the Junit tests are to get a confirmation of them working as intended. Each method within these classes is tested to ensure it is correct, and if it is correct then it adheres to the requirements.

I believe in the quality of my Junit tests because the coverage I received for each of these three features was significantly high, most were in the range of 90-100%, while only a select few did not go below a lower 80%. The only areas that show a lower coverage percentage were those that were asserting a throw, and by giving the creation of a new object variables that knowingly exceeded the character limit it does not work and gives the assertion throw. When it does give a throw, the following lines of code are never run and therefore show a less coverage percentage, but this was intentional and was operating as intended. Otherwise, my percentage for all these classes would be at a higher level.

As stated previously, all three of these features and different classes are built to be the same, so an example from one would be considered the same as it would be in another. In the example of demonstrating how I ensured my code was technically sound is in the way I ensure that the Task ID is unique when adding a new object. This is twofold, in my TaskService.java class, on line 31 there is this method  
Graphical user interface, text, application

Description automatically generated  
This is adding a new task so long as the ID returned using the findById method is null, which means it does not previously exist. The method used here starts on line 13 in the TaskService.java class  
Graphical user interface, text, application

Description automatically generated  
This uses a for loop to iterate through the array of objects, checking each index and comparing the ID values. If it does find the same ID, that ID is then returned, if not it returns null. The add Task method requires a null to be returned to ensure that the new object is having a unique ID. Writing the code in this way means that the tests only need to check for the expected output, and the methods themselves do the heavy lifting for the tests to ensure they are correct.

In the same example again, I believe my code is efficient in the way I defined the creation of variables for the object. It is done in the methods in the object classes, instead of later in the Service classes. This can be seen in any of the setters used in the three features object classes, so starting on line 16 of my Task.java for example:Text

Description automatically generated  
Here you can see the method to set the task ID, if the task ID being passed is null, or above the requirement of the maximum character limit, it throws an Illegal Argument Exception, and informs the reason why saying “Invalid Task ID”. This could have been done either via through tests, or through the Contact/Task Service class, but since it is written specifically in this method, it is more efficient and will always ensure that no variable for the ID can be set if it is given null or above the character limit.

My software testing techniques centered on one of two things, either the test was successful by passing a parameter I knew would be successful or threw an exception when I wanted it to by passing a parameter I knew would fail. I first tested to see that the creation of an object for each class with the proper parameters would pass and be set properly, as I did not test for an exception because I purposefully designed it to work correctly. I tested the variable length by expecting an exception throw by passing a parameter to each method that would be too long to be successful. And I also ran a test by ensuring that no parameter passed was null, and while I could have passed a null parameter for an assert throw, asserting that it is not null instead was easier.

For each test I performed, I would only test one or the other, but usually not both. In the example of testing a variables length to conform to the requirements, I would test to see if the variable was too long, in which case it would cause a failure or exception, but I did not also test for a successful creation of the variable by being a length less than the requirement in which it would pass. The only other thing I did not use when it comes to testing is specifically testing my created method for findById, which I used in the Service class. But by testing the other things like adding or deleting tasks, it is using this method already, so it would have been redundant.

The practical uses for any of these techniques is confirmation that the code created is running as intended. You wouldn’t want for example the parameters passed to a method be stored in the incorrect variables, by testing that the construction of an object was successful, you confirm that the parameters passed go to where they should. And the same when adhering to the requirements of certain variables not exceeding a length, these tests confirm that you cannot enter a parameter longer than those specified requirements, and it would not be accepted if it were too long or null. Using these testing techniques can help illustrate what code is being run and how it is being used, especially by coverage because you can see the length of your testing coverage and ensuring everything is running properly, and especially that everything you are testing is adhering to the requirements of whatever project you are working on.

One of the ways I had employed caution when running my software tests was double checking and asking myself if I had done all that I needed to when it came to writing the tests and ensuring I met the requirements. I was cautious not to overlook anything and there was also a time I had to double check that the creation of the ID variable for these classes was unique for the requirement, as when I had finished writing the method that does this and went to test it I was not sure if what I had created was doing what it needed to so I took extra care during this time to re-analyze my code and the tests working with this specific method. If I had not employed caution, I may have assumed that the ID was being created as unique, and it may not have been which would mean I would have failed a requirement for the project.

In a similar way as just mentioned above, there is a way that bias can come into play when reviewing code. I could have been biased and assumed that for the method I created for the ID creation was unique and I did it correctly the first time. But instead, I decided to do an extra double check on the code to ensure I was not releasing code that does not do what it needs to or does not meet certain requirements. In these situations, it is best to either limit or ignore bias altogether and run your tests to ensure everything works correctly, and not have any oversights in any specific area.

Employing caution and ignoring or limiting bias are just two things that play a part in ensuring the quality of code you are developing or testing. It is important not to take shortcuts or make assumptions that everything you created works as intended, because if you end up releasing something that has defects or obvious flaws it will cause problems for whomever the software is meant to be used by, and it could also damage your own image of being a reliable and disciplined software developer. In the example from before of double checking that the ID creation was unique and working properly, I could have taken the easy way out and simply assumed what I had was correct the first time, but I chose to dedicate myself to the quality of what I was working on and take the time to make sure I was doing everything that I needed to. Even if in the end I didn’t need to fix or change anything, I had the confirmation that what I had was correct and that gave me the peace of mind to move forward with development and testing. This also means that I am ensuring that I will be avoiding technical debt later because I didn’t let a potential possible defect escape my notice and I double checked all my work. It is far easier to fix something simple while still developing the software and testing it, then having it be discovered later and thus become more costly to fix it.