Oscar Rosa

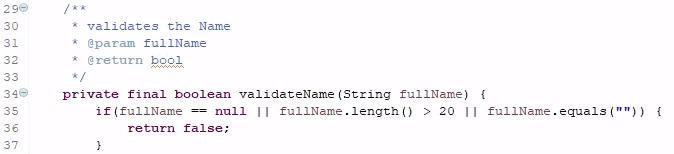
CS 320

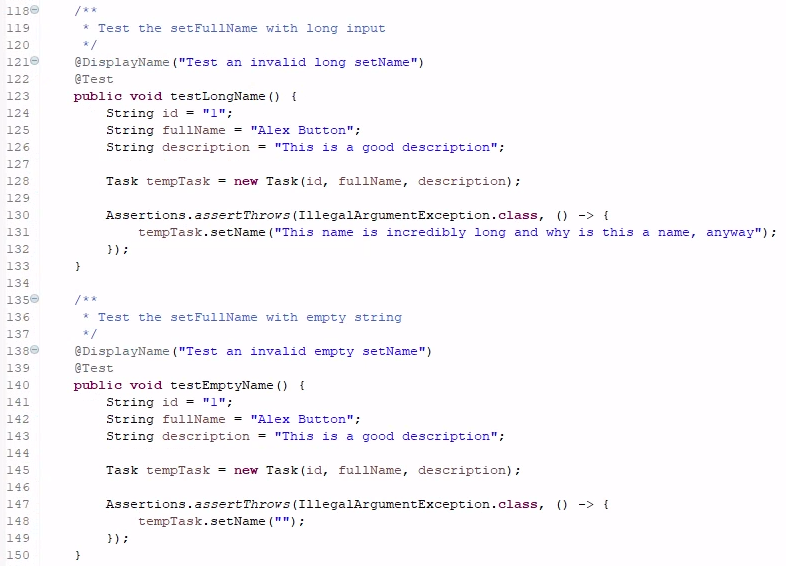
Southern New Hampshire University

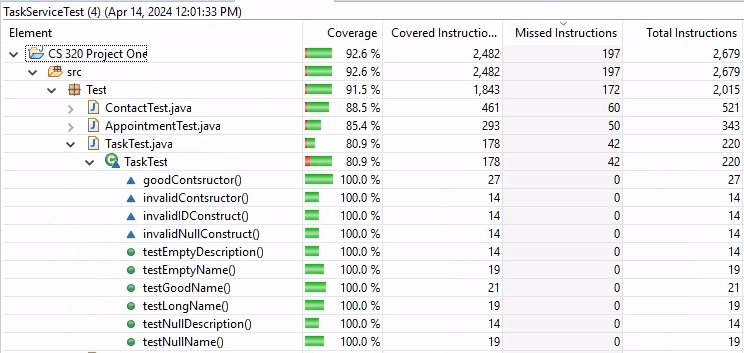
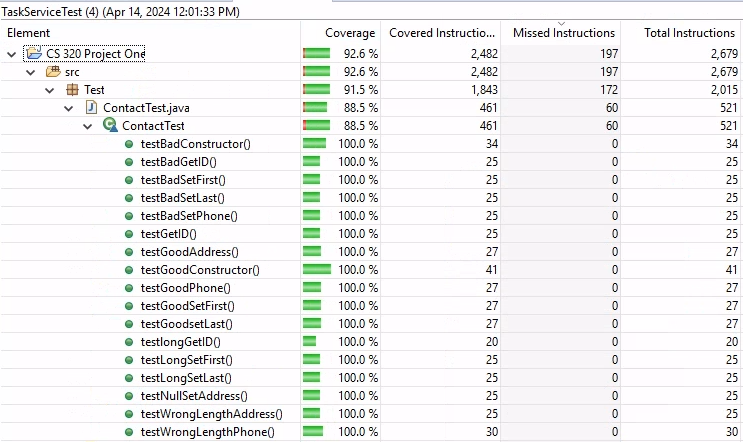
04/19/2024

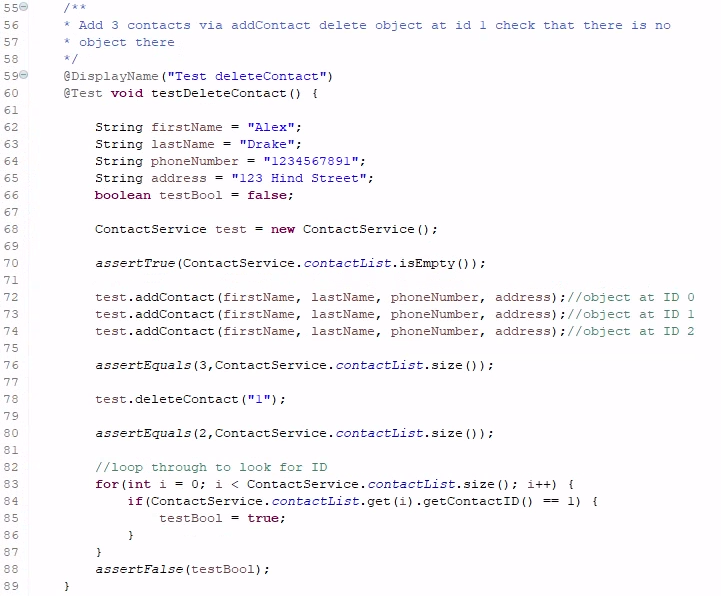
Project 2

While working on this project and creating the model and service classes as well as their subsequent tests using JUnit, I chose to adopt two mindsets while doing this. One mindset was that of a software programmer and the other was from the mindset of a malicious attacker. To that end, I utilized the unit tests to test the boundaries of these specifications, such as a specification that was given was that a full name could not be null, cannot be greater than 20 characters, and must be mandatory. To test this specification, I used no less than four branches to accomplish this task.

In addition, I tested a valid version of the name, null string, long string, and empty string, and have some samples of that provided below.

Furthermore, I did not try for complete 100% coverage of both the Task and Contact classes as the constructors had roughly +3 parameters to set. While testing the code I did not believe it to be worthwhile to test good and bad constructors for each of the parameters, as instead I chose to focus on the getters and setters with only a good constructor and a single invalid constructor. Images for both the Task and Contact class test coverage can be seen below.

 To ensure that my software was technically sound, I did my best to ”test to the spirit of the code” so to speak. An example of this can be seen in my ContactService file in which it required a method to delete an ID. This method could have been tested in a multitude of ways, one being to add three objects to the collection, delete one, and then confirm the collection was now the size of two, and the second would have been to look through the entirety of the collection to locate what was deleted to confirm that it was gone. The issue with only doing one of these methods is that it leaves room for aberrant behavior. Meaning, if only the collection size was checked then the wrong item could have been deleted and the test passes. If the item is searched for as the only technique, then the test could pass if all the items were deleted accidentally. With that in mind, I tried to test for both and an example of this can be seen in the image provided below.

As mentioned previously, it is not enough to just check the size of the collection once something has been deleted, but it is also equally important to check that item is in fact deleted as well. In addition, to take this a step further would be to ensure that the remaining objects are still intact and as expected, in this instance I did not include this as an item to test for in my code. However, I did test to make sure that objects were instantiated only when requested which can be seen in the image below.



Furthermore, I also checked that after a failed string updates that the string was still as before the attempt which can be seen here:

 While working on this project and the 3 modules needed for it, I employed the use of unit testing and static testing, both of which are a form of whitebox testing. With static testing, it involves studying the code and comparing it to the specifications to identify bugs or other deficiencies that may be present within the code. For me I used this technique to find and fix the offending code after the code had failed one of my JUnit tests. As I was writing the class methods, I was also attempting to design the logic to match the specification document the project was required to have. During the unit test creation, I re-read the specification document and tried to find ways in which to prove that I had made errors during the coding process. On a few occasions one of the JUnit tests failed and I had to inspect the static code to figure out why it had failed. In one such instance, I had noticed that I had used the wrong attribute within the logic branch I created, so because of that mistake was the reason my code failed the test. In addition, I learned that just merely making sure that exceptions being thrown when expected was not enough, but catching the correct and expected exception is also important. For instance, in another of my cases I was trying to assert that a constructor of mine would fail due to bad input. Because of this an error was thrown but not the error that was expected, so in a subsequent code I began evaluating the exception type to ensure that it was the intended exception I was expecting and that my tests were not creating false positives.

The service classes in this project had very little system testing, and no integration testing was done to it. The specifications created a model class and service class that drove and interacted with the model class. I believe that you could test the model class through the service class, and you would be able to treat this interaction as a system. To that end, I did a bit of this in a later developed code, however, this missing testing should be treated as a technical debt.

With integration testing, this would test the entire application which brings all the systems together for testing. In this instance, the system and integration tests would be the same, with that in mind I would have been able to validate the TaskService, AppointmentService and any other classes to ensure that all would work together effectively and efficiently in the integration layer, however this was not done while working on the module assignments. In addition, I did not conduct automated testing as all my JUnit tests that I created were manually ran. However, I could have created a service that would have been able to run the JUnit tests after each build-event within Eclipse, but this would have been more important for large applications as opposed to what was required in this project.

Furthermore, there was no security scanning performed on this project. The libraries and components of the program need to be scanned for security vulnerabilities. Security testing is quite important, especially when dealing with data. Currently, there were no databases or user inputs into the system, therefore protecting against improper formatted inputs such as SQL injections, or buffer overruns was not needed for this project. Unit testing is always beneficial and can help to ensure that small issues discovered early on do not become huge problems later. Unit tests can also provide aid in strengthening code against missed logic branches, such as denoting what the value is for null, missed specifications, or even maximum string length for characters. In addition, system and integration tests are also useful as the code base grows and more interconnected components become introduced then the more important these types of tests become.

The mindset I had for each phase of the project was slightly different as I previously mentioned in the beginning. In the coding phase I tried to interpret the true intentions of the requirements and not just what was written down, such as with the task of “the appointment object shall have a required description string field that cannot be longer than 50 characters. The description field shall not be null.” Put simply, this means that there should be a logic test for the length and if the passed argument is null. However, the spirit of the requirements would suggest that there needs to be a description. It would stand to reason then, that the description should also not be blank/empty, therefor the validation test would be as follows:

A close-up of a computer screen

Description automatically generated

Furthermore, when looking at how the service classes are intended to be used with the model classes I determined that using a hash map was the more efficient route than a list would be. Instead of creating code that searches the collection for an index, I simply used the hash map method to achieve those results and by utilizing hash map also simplified the JUnit code.

Use of list collection in remove method (loops through entire collection):

A close-up of a contact list

Description automatically generated

Use of hash map (no loop):

A close-up of a computer screen

Description automatically generated

While working on this project the main bias I tried to remove or at the very least test for was that of “I know I wrote my code correctly”. From this I learned that conducting tests does not break your code, rather it breaks the illusion about the quality of the code. With that in mind, instead of just testing that any exceptions were thrown, I would utilize JUnit tests to check and ensure that the correct exception was thrown as seen below.

A close-up of a computer screen

Description automatically generated

“Technical debt happens when you take shortcuts in writing your code so that you achieve your goal faster, but at the cost of uglier, harder to maintain code. It’s called technical debt because it’s like taking out a loan. You can accomplish more today than you normally could, but you end up paying a higher cost later” (Technical Debt, 2024). Avoiding technical debt is a long process and definitely not a short one. This idea has to be revisited at regular intervals. Security issues and vulnerabilities arise every day with exploits being attempted in just mere hours. Furthermore, technical debt is not always a bug or deficiency as it might be a missed use case. “Technical debt describes the consequences of software development actions that intentionally or unintentionally prioritize client value and/or project constraints such as delivery deadlines, over more technical implementation, and design considerations…” (Technical Debt, 2024).

All in all, beyond the obvious benefits that JUnit tests provide, they can also serve as indicators for code quality. Often times, more mature software has higher code coverage which can lead to better sustainability, whereas low code quality should be treated as technical debt. Within this project the JUnit tests provided a “second set of eyes” so to speak and have captured a few cases of improper copy/paste. In order to get the most benefit from unit tests it is imperative that developers and testers put aside their pride to ensure what is actually intended for their code occurs along with ensuring that no adverse side effects occur alongside it.

**References**

**Technical Debt. (2024). ProductPlan. https://www.productplan.com/glossary/technical-debt/**