# Project Two

The testing approach that I used for this project was almost exactly aligned to the software requirements. The tests that I created in both scenarios were derived from the requirements given, and that also helped in the design of the underlying classes/code being tested. In the Contact Service requirement, I chose to implement the classes first, based on the requirements, and then wrote the tests to make sure those requirements are constantly checked. For the Task Service requirement, I took another approach which was to write the tests first, and then write the classes. By doing so, not only was I able to write the tests without thinking too much about the implementation, and only focus on the requirements, but it also helped to design what fields and methods I would need when implementing the underlying classes. The tests would start with compilation errors, which would let me know which methods I still needed to create. Once all the compilation errors were removed from the test file, I know that I've fully implemented the class, at least for the purpose of testing the requirements given.

As far as code coverage is concerned, I believe the tests I wrote were of high quality and very effective, since I was able to achieve 100% code coverage on each of my classes, with all of the tests I wrote passing. In the process of writing them, it also helped to show the importance of checking not only the "happy path" of making sure things work when correct data is entered, but also the "not-so-happy path", in which incorrect data that we wouldn't expect is entered. When only writing tests for the first case, I noticed that my code coverage was not yet at 100%, since the parts of my code that are trying to check for this incorrect data were never tested. After writing those tests, as well, the code coverage increased to 100%, and it gave me more confidence in the code, knowing that incorrect data is also being handled appropriately.

One example of ensuring my code was technically sound was the use of a "count" method in both the ContactService and TaskService classes. Both classes make use of a HashMap to store the items internally, and so I wrote a custom method (line 42 in TaskService.java) to return the current size of the HashMap. Then, in my TaskServiceTest.java class, I make use of that method to ensure certain processes are performing accurately. In the "testAddTask" function (lines 41-43 in TaskServiceTest.java), I make sure of that count method twice, once before adding a new task, and once after, to ensure that the task truly was entered into the HashMap.

To make sure the code was efficient, I made use of the "@BeforeAll" annotation provided by JUnit. By writing a function with this annotation, rather than the regular "@Test" annotation, it would run the function once, before all of the other tests, and persist that data until all of the tests have completed. By doing so, I was able to make sure of a "test contact" and "test task" object (line 21-28 in TestServiceTest.java), without having to copy/paste the details of creating that object or having to call that function at the start of each of my tests. In this example, it wasn't that much of a difference, but if we had to make a connection/request to a database for retrieving this data rather than creating it manually in the program, it would be much more efficient to use this method.

For this project, I employed three of the following testing techniques: Static analysis, black-box testing, and a minor version of white-box testing. For the static analysis, I made use of myself as a resource, by manually checking the code over, which was not too difficult with the size of each codebase. Besides that, we also made use of JUnit, a popular testing library within Java, that allows us to write static tests against our code, which will tell us immediately upon running our code if any of those tests fail. I also used a form of black-box testing by writing each of those tests, as those tests were derived directly from the user's requirements, so assuming all our tests pass, that should theoretically mean that all of the user's requirements have been satisfied. Each of those tests are not concerned with the internal structure of the program, but rather if the user's requirements have been fulfilled. Lastly, I used a form of white-box testing, to ensure that a certain piece of the internal structure of my code was working as intended. In my case, the HashMap that held the various objects in their respective "Service" objects, I created a "count" method which returns that current size of the HashMap at a particular time. That way, during some of my tests, I can ensure that data was entered/deleted properly by checking that count.

Not every testing technique was employed during this project. The two main techniques which were not used were a "Code Review" (a form of static analysis), and dynamic testing. For a code review, one or more other members of a team (or anybody familiar with code, really) would look at our codebase and determine if they see any issues with it. This can be incredibly beneficial, as having someone with "fresh eyes", or someone that hasn't been working on and staring at the problem for a long time, look at the code, they might be able to point out issues that the original developer overlooks. However, in the environment we're working in, we don't necessarily have another person to review our code for us (except for our professor) and so I wouldn't consider this to be a technique that was used. The other method that wasn't used was a form of "dynamic testing". I think this was mostly due to the nature of the code itself, as it is not complex enough to need to be dynamically tested. As more elements are added to the codebase, and the complexity of the classes we've written increases, dynamic testing might be something we employ in the future.

Out of the techniques discussed above, I believe static analysis and testing are the most beneficial for different software projects and situations. No matter the project, the need for requirements will always be necessary, otherwise, we'll be building software for no reason and with no goal. Once those requirements are nailed down, it's still necessary to scrutinize those requirements before any coding happens as much as possible. By doing so, the cost and time for developing the software, and the number of bugs generated by writing the code, can be greatly reduced. Then, once development has started, using a testing library (such as JUnit) is another highly important technique to employ in any project. Without doing so, we can't be sure that certain aspects of our code are working as we intend, we can only hope that they are. By writing these tests, we can instantly see if any of our assumptions are found incorrect when writing new code, and it also gives us a greater amount of confidence in the software itself.

While working on this project, there was certainly a level of caution gained by writing tests for each of the different classes. By writing the tests, it showed how the code should be used, and how the different classes interact with one another, while also satisfying as much as possible the requirement given by the user/client. By writing the tests up front, it can often lead to the creation of other tests, without the burden of how the underlying code might be implemented, which can help ensure the code will be working as intended once it is implemented, while also providing a roadmap for the developer in the form of the unimplemented code.

Bias will exist to some extent, whether conscious or not, especially for a developer who has spent a lot of time working on a particular project, no matter how much they try. Often, they only consider the “positive” paths, and might not necessarily consider checking the opposite case during their testing. Having code reviews, or simply having someone else check over your work, can lead to additional tests that the original developer might not have considered.

As software engineers, if we are disciplined with the techniques we learned during this project, we will be able to create very robust software in the future and honing this skill will allow us to have this confidence in the software we build in a quicker fashion. The way I try to avoid technical debt is tackling it right away. It’s not always feasible but doing so makes sure the issues aren’t left behind.