Southern New Hampshire University

7-2 Project Two Submission

John Price

CS-320-10372-M01

Omar Toledo Lopez

For this assignment, we were required to create six classes broken up into three sections of two. There is a class that creates the class object, which holds the relevant data, and then an additional class with the same name but with the word “service” appended to it that handles the subsequent programmatic operations. The three classes are Contact, Task, and Appointment, and their sister-classes are called ContactService, TaskService, and AppointmentService. The testing approach that I took for this assignment was very closely aligned with the software requirements as well as the suggested approach from the professor in one of their announcements, where they listed the most important things to test for in a program. In order to ensure that the software requirements were met, I made sure that my test checked to see if the requirements were met when the correct information was provided, but also that specific tests failed when the incorrect information was provided. This approach forced me to write two tests for almost every requirement to make sure that the correct method works and that the incorrect method doesn’t work. This is very important because often code will work the way the developer wants, but also in ways they don’t want it to, so making sure that tests fail when they are supposed to fail is vital. Being this thorough while testing is taxing, but worth it overall because it helps improve the test coverage and effectiveness of the testing suite.

When I was writing the JUNIT tests for this assignment, I found myself enjoying it more than I thought I would because IntelliJ was able to fill in gaps where I was using similar code. One thing that I did to make sure that my tests were technically sound was to make sure that when my tests failed that they triggered a throw call from the class function. I didn’t want my tests to just fail and not know why they failed or where they failed, so I made sure to implement exceptions to be thrown in the code, which makes it much easier to find mistakes. One specific example is when I’m testing that multiple tasks being added with the same ID will fail because it would create a duplicate ID in the data, so I made sure that it throws an illegal argument exception.

The software testing techniques that I used for this project were mostly unit testing because the system was relatively small, and integration testing because there were still multiple components interacting with each other. The unit testing that I did mostly involved making sure that the data was being structured correctly and accurately. This was done for each of the three main classes, Contact, Task, and Appointment, because most of these classes’ utility was to construct the class object with the data provided. A specific example of this would be in the testing for the Appointment class; there is a test that is supposed to fail because it is trying to create an Appointment object with an appointment date that is set in the past. This obviously should not be allowed, which is why the test is designed to fail, but doing this makes certain that there won’t be appointment objects with invalid dates. The integration testing that I did was in the testing for the service classes, ContactService, TaskService, and AppointmentService. These classes handled the programmatic mutation of data with class methods and need to be able to interact with different components to work correctly, most importantly, their main class. Testing these classes often required each test to create a class object from the main class to use as sample data, but confirming that these components can easily interact with each other is vital for integration testing because oftentimes the introduction of new components can lead to old logic breaking or failing. One specific example for this would be in the testing file for the TaskService class, where there is a test that is designed to pass, and tests that allow you can add multiple tasks at the same time. The test has to call the main class’s constructor multiple times and the methods in its own class, which makes it very important that these components can operate together.

A software testing technique that I didn’t use for this project is system testing because I didn’t need to for something this small. Usually, system testing would be done after integration testing, once all of the components have been fully integrated into the system, the integrated components would then be used as input for the system testing. Since this project was relatively small, the need for something like system testing doesn’t arise because the system is simple in design.

My mindset for this project was focused and determined because I put myself into the position of a software engineer who has been tasked with important work. Since I underwent this assignment with the mentality of an employee, it was paramount that I proceeded with caution, especially when writing the tests for this project. This is because the testing is the last thing that is preventing bad code from either reaching the production pipeline or a supervisor who will then need to correct the issues. Using caution is vital during the testing phase, but also important when working on the bulk of the code, especially when considering things like the complexity of an algorithm. Code doesn’t always need to be the most efficient complexity possible, but avoiding unnecessary bottlenecks is massively important because they can build up and cause a program to drag to a standstill. For example, if a function is already looping through a data structure and there is a nested data structure inside that needs to be iterated over as well, the developer needs to seriously consider if that is necessary, but the complexity of the algorithm will increase if they have a loop within a loop.

Avoiding bias as a human is fairly impossible, but that is why it is so important to try as hard as possible. It’s so easy to write a bunch of tests and then realize there’s a better way to do something, but decide not to change anything because the original code still gets the job done. Deciding to go back and force yourself to do the work actively makes you a better developer and strengthens your resolve. Specifically, if a developer were to be responsible for their own testing and choose to be lazy since nobody is checking their work, it could cause serious production issues. If that code has bugs and breaks things when integrated into the system, it is the developer’s responsibility to find and fix those issues, and if they aren’t able to, then they aren’t able to perform their job.

Discipline is a skill that is earned and built over time, like hammering metal until its form is shaped into a blade. As a software developer working in the tech industry, there is always new software and devices that can help skip hard work, but oftentimes these lead to inexperienced developers who aren’t able to work independently. This is why discipline is important for developers and why cutting corners during testing always creates negative results. It’s always better to have unnecessarily extensive testing because it creates peace of mind and allows the developer the freedom to work without fear of breaking other people’s work. Unfortunately, there’s no easy way to combat this issue, so we’re forced to simply hammer the metal every day, slowly building the discipline that makes us great. My plan is to do just that, keep working at it every chance I get, and eventually, worrying about tech debt will seem like a problem of the past.