7-2 Project Two: Summary and Reflections Report

James Ehrenberg

James.ehrenberg@snhu.edu

Southern New Hampshire University

2024-04-20

During project one we were asked to deliver three different services with unit tests for each applicable service and its requirements. The three services needed were a contact service, task service, and appointment service. The contact service requirements were for a contact class with an id field no longer than 10 characters and cannot be null. A first name field no longer than 10 characters that cannot be null. Last name, and phone number fields sharing the same requirements as first name, and an address field that cannot have more than 30 characters and cannot be null. After creating the contact class, we needed to create a contact service that would store the list of contacts. This service could also delete contacts by id and update any field other than the id field based on the provided contact id.

The task requirements were the same as contact; however, the fields were id, name, and description. The id requirements stayed the same, but the name field was increased to 20 characters and the description field increased to 50 characters. The task service could also store a list of tasks, delete tasks by id, and update either the name or description of a task by its id.

Lastly, was the appointment service. This service required an appointment class again with an id that had the same requirements as the previous two classes. Next was a date field that could not be set in the past and could not be null. As well as a description field that could not be longer than 50 characters and could not be null. The appointment service could then store a list of appointments and delete an appointment by its id.

For me to meet these requirements in the first project, I made sure my unit tests thoroughly tested each requirement of the software. For example, starting with the contact class I made validator methods to test the inputs before adding the value to the field. The validateName(String name, String fieldName) method was used to check if the value provided was either null or longer than 10 characters. If it was, then the software would throw an exception informing the user that the field was not valid. I continued these validator methods throughout each field within each class that had fields with similar requirements. After making these validators I was able to develop my unit tests around these exceptions and test that errors were properly thrown under the right circumstances. To ensure adequate test coverage, I tested everything from creation, to length, to errors and null values on every field within each class. This ensured users would get the proper messaging when entering a null value or a value outside the parameters allowed for the field.

Methods such as *testInvalidLastNameLength* on line 39 of the ContactTest file, as well as *testNullAppointmentDate* on line 41 of the AppointmentTest file were used to make sure both length and null values were tested for each class. Methods like *testAddAppointment* on line 13 of the AppointmentServiceTest file started by creating a new appointment service, a new date in the future, and a standard description. I then created a new appointment and added it into the appointment service on lines 15 – 19. Then, on lines 21 – 24 I tested to ensure the appointment was not null, that the id of the appointment was 10 characters in length, and that the date and description of the appointment inside the service matched the ones generated in the code above. This ensures the unit tests efficiently cover all aspects of the appointment class and the appointment service class.

To follow this up, methods such as *testDeleteAppointment* and *testDeleteInvalidAppointment* in the AppointmentServiceTest class, and *testUpdateContact\_InvalidField* or *testUpdateContact\_NonexistentId* in the ContactServiceTest class help to demonstrate the technicality of the tests through testing proper errors. These methods all show how we can test not only to make sure that our code properly adds new class instances to a service, but also test that our error handling is just as sound. This helps provide a more quality user experience to the end user when using our application.

During this project, beyond unit testing, I used both static testing and systems testing. The static testing was used within my IDE by showing me when a call to a class was misspelled, or a method on a class did not exist. Code completion is another form of static testing by ensuring that only existing methods and properties within classes are used. Static testing was also done on things such as my pom.xml file to ensure all packages were up to date and did not contain vulnerabilities. This kind of testing is very important to ensure the product is not shipped with something that could be exploited by a hacker and put the user at risk.

Beyond that of unit testing and static testing, I also conducted integration testing through testing the services as well. After creating each individual class, I needed to ensure those classes then integrated properly with the service classes. This was done by testing that the services properly held the list of data and could be manipulated as intended.

The systems testing, which is when we run through the program to ensure that the methods and classes are functioning properly, was done through our unit tests. After writing each unit test, I executed the test on my machine to ensure that not only did the program still ran successfully, but also that the test passed and validated the information intended to be validated in that given test.

Other forms of software testing include code reviews. Code reviews are when other developers review code written by their peers. This fresh perspective helps identify potential issues, logical flaws, or areas that don't align with coding best practices. Code reviews also promote knowledge sharing and improve code maintainability. Integration testing comes after some of the more standard testing methods like static testing, unit testing, and code reviews. Once individual units are tested, they are integrated to form larger modules. Integration testing ensures these modules interact correctly with each other.

All these different testing methods can be used across a wide range of development projects. For example, in my own endeavors as a web developer, I use unit testing, systems testing, and integration testing constantly to test both the front-end and back-end systems. Unit tests can also be used to test pieces of UI to ensure that the proper views are displayed after a given interaction. Unit tests can also be used with mock data on APIs to ensure that proper responses are returned from HTTP calls to a specific server route.

For myself personally, my mindset was already very adept in the methods of testing. I have been developing software for a long time and TDD (Test Driven Development) has been my primary form of development for a long time. As a solo developer that does not have a QA team, I must be the QA team myself. Which means I must be rigorous when writing tests and strive for as much test coverage as possible including edge cases. This helps to ensure that my code is sound and can ship bug free to my clients. As for bias, I try to always do research and learn as much as possible from other developers and from the community so that I do not fall victim to a narrow mindset.

When testing code, it can be difficult to think of every use case since as the developer you tend to think of how it is meant to be used. To avoid this, I try and have my girlfriend, or a family member run through early versions of tools that I write to see where the flaws might be. I also try and actively exploit my code or intentionally break it to see if I can find a bug or an edge case I did not think of originally. All these methods combine help to ensure discipline and thoroughness throughout my tests and help to ship the soundest product I can.