Ryan Hoskins

April 19th, 2024

CS-320

Dr. Tuft

Project Two – Journal

When testing any software or code used in an application, it is important to reference specific software requirements that have been established as priorities. By beginning the testing process with an understanding of the requirements, testers can formulate tests that ensure specific aspects of the platform are properly coded and behave as intended. For these reasons, my approach in each of the three features was heavily influenced by the software requirements. This can be confirmed by reviewing the code for each feature, where each of the requirements was thoroughly tested. As an example, the appointment class and appointment service file had several unique requirements, such as the ability for the application to ensure an appointment cannot be scheduled for a past date. My testing approach took this requirement, among others, and expanded on them to ensure any potential scenarios could be addressed. The AppointmentService.java file was tested to ensure any updates to the appointment date would be valid, as seen here:

A screen shot of a computer code

Description automatically generated

This demonstrates that the program will be tested to ensure the date cannot be updated to a past date, that the format is correct, and to ensure invalid characters cannot be used. Additionally, the Appointment.java file was tested to ensure the Constructor would abide by the requirements:  
A computer screen shot of a program

Description automatically generated

Through all these Unit Tests, the requirements are thoroughly abided by to ensure the application functions as intended.

Considering the requirements for each of the components of the application, the JUnit tests created provide a high percent coverage to ensure much of the code used in the application was tested. By achieving a high coverage percentage, the unit tests created demonstrate the application was thoroughly tested. As a whole, the entire application and all its components resulted in a coverage percentage of 83.7%, as illustrated here:  
A screenshot of a computer

Description automatically generated

By achieving a coverage percentage of greater than 80%, the Junit tests can be confirmed as effective in their performance and ability to thoroughly test the code used in the application and its components.

The process of creating JUnit Tests presented some challenges, as I had to determine the correct way to test a part of the code. Even if I knew the code functioned as it should have, I needed to write a test that would demonstrate it does. As demonstrated in a previous screenshot, there was a great deal of effort put into the code to ensure it was sound. To ensure the code used was technically sound, I reviewed all requirements and made sure to write tests that would cover a wide variety of scenarios. Additionally, I wanted to make sure invalid entries used for an object would not be accepted, whether in the creation of the object or updating of the object. For example,  
A screenshot of a computer code

Description automatically generated

This screenshot demonstrates my testing strategy to ensure an entered phone number would not include an invalid character, length, and to ensure it is not left empty. To further demonstrate, the application utilizes technically sound code to ensure this as well, as the application will perform an initial check to verify the field isn’t empty or of any length other than ten characters, and then iterate through the string to ensure each character is a digit:  
A screen shot of a computer code

Description automatically generated

Additionally, efficiency was also prioritized in the testing process. This can be demonstrated by reviewing some of the tests used. For example, in the AppointmentServiceTest.java file, this test was created to ensure it could add multiple contacts:A screenshot of a computer program

Description automatically generated

This test demonstrates no branches were missed and we called directly upon the createAppointment method using passed information. By creating the appointments, calling of the method to get an appointment, and asserting they are not null, I’m able to demonstrate a streamlined test that tests two methods and proves the created Appointments are not null.

Throughout the creation of the application and its various components, several testing techniques were employed. Techniques include boundary value analysis, input partitioning, exception handling, Decision Testing, and manual review were all utilized throughout the code writing and testing process. To ensure an Appointment ID, Contact ID, and Task ID had a valid length of characters, boundary value analysis was used to ensure an ID would be valid if it were ten characters, less than ten characters, greater than ten characters, and empty. This also creates a partition of one to ten characters, inclusive. Exception handling was considered, as it is crucial for an application to appropriately handle scenarios and to relay information that is helpful to understand what happened. For this reason, I included error handling that would explain why the creation, removal, or update of an object failed. The testing and code used in the application also used Decision Testing through the Update methods in each component. Since the application needs to be able to demonstrate it can update the appropriate information, I implemented a Switch Case that handled each of the scenarios for a request to update any particular valid field, and would handle any issues resulting from each. I also included a Default case to make sure the user enters a valid option. Manual review was a big part of the code creation and testing process, which involved me carefully looking through each line of code to ensure the flow of actions were logical and consistent. I also manually created objects, removing, and updating them. Lastly, I also created a visualization of the program and its order of operations through each process.

The testing techniques used in the creation of the project helped ensure an efficient and technically sound product, but there were other testing techniques that I did not utilize. Some of these include Decision Tables, black-box testing, and integration testing. Decision Tables are typically helpful for any actions or process that involves multiple choices made, each with a unique end result that needs to be achieved at the end. In this particular program, there is no such need for that. Integration testing can be used when a developer or tester needs to ensure larger updates or aspects of a platform can be implemented into a larger system and everything still function as intended. This can be a critical form of testing for platforms or systems using several modules of code that need to run seamlessly together. This can be a very important form of testing when APIs are used or data is transferred, as those projects typically rely on many separate, but combined, components that need to interact with one another. Another form of testing that was not utilized for the most part is performance testing. Although I did choose to use a HashMap as the internal data structure for performance reasons, there was not any reason to utilize performance testing for this project. Performance testing can be important for larger systems that must scale efficiently or that are expected to experience heavy traffic with many interactions and tasks throughout.

These techniques each offer value to a tester or developer working on a project, although some may offer more value for specific types of projects while others are best used at different stages. For example, Integration Testing, as mentioned earlier, is essential for larger platforms using many modules of code. However, integration testing by nature needs to be done after the initial code is created and tested. Additionally, Integration Testing may not be the best option or feasible for projects that are not expected to need to integrate their code into a larger system. Unit Testing can be done often. Unit Tests ensure a specific method or piece of code functions as it should, which is why it is important for them to be used early in the development process. They offer a lot of value, as they are typically able to catch issues early and offer the opportunity to quickly correct any issues before it is too late or expensive to do. However, larger system will likely have limitations in the amount of unit tests that can be used. It may not be feasible to perform unit tests on every bit of code used in a massive system.

The mindset I used throughout the project helped ensure caution was taken. I wanted to ensure that user input was properly validated in more ways than what was even mentioned in the requirements. For example, when creating an Appointment object, I wanted to ensure the date of the appointment utilize the modern Java option of LocalDate instead of the outdated and potentially dangerous predecessor. Additionally, I ensured the date would be parsed and validated:  
A screen shot of a computer program

Description automatically generated

This screenshot demonstrates an understanding of caution and how relationship of the various components of code used throughout the project. By ensuring the date is parsed, the program can use the information in a consistent format for testing. A project involves many components of code that are expected to work cohesively together in an expected and consistent manner. For this reason, it’s important to recognize how complex a project can become, as there are many scenarios in which the program could find itself in and ways in could be used that would not have initially been expected.

For this very reason, it is important to limit any bias going into the testing stage of a project. It is common for a developer that coded the project to go into a testing stage with certain assumptions of false confidence. Due to this potential pitfall, I attempted to go into testing as objectively as possible. When reviewing what I should test, I did not simply stick to the requirements provided, but instead analyzed the code itself. For example, a Constructor will be passed data that was input for various attributes, so I wanted to make sure the program had a way to ensure any of those fields were valid when creating, updating, or removing an object. While I could simply test the Constructor and assume the add, set, get, and validate methods would function as intended, I tried to limit any bias by testing each of these. This can also be seen when attempting to test one of the fields in the Appointment class:  
A screen shot of a computer code

Description automatically generated

This screenshot demonstrates my effort to remove a bias. Instead of assuming the formatting would be valid based on the code used in the Appointment.java file, I made sure to create Unit Tests to ensure the appointment date cannot be null, made for a past date, has the proper formatting, and does not include an invalid character.

This same example can be used to demonstrate my commitment to quality. Developers cutting corners and the impact those decisions have on various projects can be seen in many devastating examples that are costly down the road. When approaching this project, I attempted to abide by a level of discipline I would be proud of and to ensure I created a quality of code that I was confident in. For example, cutting corners when testing could present itself in the form of just testing a single instance of creating an object, but not doing so in the various scenarios that could present a problem. For this reason, I tested the creation of an object in many different ways, varying the conditions slightly between each. I also worked hard to create sound code. For example, when creating the Constructor for the Appointment object:  
A screen shot of a computer code

Description automatically generated

One of the requirements for the Appointment class is that the Appointment ID cannot be changed. For this reason, I did not include a setter for the ID field, but I wanted to ensure it would still be validated when an object is created. Each of the other fields are set to the result of passing the input through the validating methods. This ensures that when an object is created, each field is properly validated. Additionally, for the code I created to update an object, I wanted to ensure the user entered valid options. To make sure this was accomplished, I first used an if-else that checks if the entered appointment ID is present in the HashMap. If it is, the application moves onto a Switch Case based on the field the user entered, where each case calls on the setter of that field with the passed new value. This ensures each step is appropriately validated. I also thoroughly tested the Update method:  
A screenshot of a computer code

Description automatically generated

These are a few examples of the ways in which I tested the Update method. Not only did I try testing the update method with an invalid new value, but I also tested to ensure the method could correctly handle a scenario in which the user inputs a nonexistent ID or an invalid field.

Moving forward, I plan to incorporate these techniques into any testing I perform. It’s essential to go into any testing with an objective and unbiased mindset, as it can be very easy to operate under assumptions if you are testing the code, you also wrote as well. For that reason, I will analyze any applications I work on to review the various components used in it, how each interacts with other components, and then be sure to check each interaction. I will also carefully review any provided requirements to reference during the initial writing and testing stages.