Dante Lee

13 April 2023

CS-320-T4206

Project 2 - Summary and Reflections Report

Summary

Throughout the development of each of the three features in this project: Contact, Task, and Appointment classes and services, there were many considerations that had to be taken both for the implementation and for unit testing strategies. In order to fulfill all software requirements, it was important to first lay the groundwork with the class structure for each object. This involved which fields would be stored in each occurrence of a Task, Contact, or Appointment. Along with these, there had to be some input validation when constructing an object, to align with the field requirements such as character limits, data types, and other formatting considerations like mutability. For example, id fields were set as final to disallow editing or overwriting their initial value. Since these checks are all done at the time of construction, the integrity of any data stored is protected. Then the accessor and mutator methods were implemented, which were straightforward, however the same input validation techniques had to be used on these methods in order to avoid someone first being able to create a valid instance, and then mutate one of its fields to something illegal. This approach was not taken at first, but upon further investigation the program proved to be vulnerable to that happening. As for the immutable id field, we did not create a public mutator method for it at all, since it was to be final and unchanged.

As for the service classes for each, these are where the data structure is created in memory for storing, in my case, a hash map containing all instances of an object retrievable by id. For testing purposes, additional methods were created on top of adding objects to the list, removing, and updating. These additional methods include returning a specific instance of an object in memory by id, so that their fields and statuses could be analyzed upon changing or removal, and a method for returning the size of the list so that the add and remove functions could be verified for correctness as well.

The unit tests that I created for each of the features, I wanted to achieve 100% code coverage, so it was important to test every function in a success and fail state. First, the constructor can be tested by attempting to create a new instance of each object with a single field violation present. Most of the fields used are strings, so the input can be anything as long as it is not null and not over the field length limit. I first test against each field individually being too long, and then for each of them being null. It was verified that each test threw the anticipated IllegalArgumentException in these cases. The field that was not a string, the appointment date field, had to be a future date, so it was checked that the date occurred after the present moment. Next, each of the data object accessor and mutator methods were verified by creating a new instance with each respective field (one per test case) being constructed to an initial state, then modified with the mutator method, and then verified that the change occurred by using the accessor method. This assertion covers the mutators' functionality, while also covering the accessor method, making for an efficient, yet compartmentalized test case strategy.

Some requirements for the services of each feature were that no objects with duplicate id could be added to the data structure, so in the service method for adding an object, the id of the object was searched for in the hash map, throwing an IllegalArgumentException if present. The inverse is true for updating a contact, where the id would be searched, and if not found, would throw the same exception because we cannot update a contact that doesn't exist. In the test cases for the services, these features were verified by attempting to add two objects with duplicate id, and the correct assertions were thrown. It was also tested for updating a contact with a non-existent id, throwing the same exception. Then, we used the additional methods mentioned earlier, and created a new instance of the service structure, checked that its length was 0, added an object, checked for size 1, deleted an object, and again checked for size 0. Another test case was adding an object and then updating its fields, then using the method to access a specific object in memory by id, in combination with the accessor method to verify that the correct field was updated to the correct new state. This showed that not only did the add, remove, and update methods work properly, but the extra methods did as well.

By conducting thorough test cases like these, testing every method, and creating new methods to deeper validate the functionality that was implemented in this design, we achieved 100% code coverage for every feature.

Reflection

The testing techniques used in this project are explained in depth above in specific detail, along with their effectiveness and relevance to the software requirements. Other techniques that were not used in this project include static testing, which would identify any dependencies or vulnerabilities in the code. This was not so necessary, as this is all base Java with only one additional package used for date handling, which can be kept up to date and will be maintained by official channels. There are two more relevant testing techniques to consider for this project:

Integration testing involves testing how different components of an application work together, and is essential for identifying problems in the interactions between different components of an application. It is useful for large-scale software projects with multiple modules or components, ensuring that all the components are working together as intended and can help to prevent integration issues down the line. In this project, we did not integrate much of the features with each other, aside from objects and services.

Acceptance testing involves testing an application's functionality against the requirements of the customer or end-user, and helps to ensure that the software is useful and meets the needs of the intended audience. In unit testing, we practically achieved this because the implementation met the requirements that were put on it, but the end user did not interact with it in this state. This code would be dressed up with a more user friendly graphical interface.

Bias when reviewing code is a real thing, and can occur when a developer implicitly believes that their code has less errors since they were the one who made it and witnessed themselves attempt to include security and validation measures. I tried to limit my bias in reviewing and writing test cases for my code, but I ran into one problem. When I added the input validation into the constructor methods, I overlooked the need for them to exist also in the mutator methods. If I were reviewing someone else's code, I would have probably scrutinized more due to it being a clean slate that I hadn't laid my eyes on before.

Discipline is also a hugely relevant factor, and plays a role in writing test cases that not only achieve 100% or close code coverage, but keeping test cases compartmentalized and focused on one small feature. There can be intuitive ways to coordinate more than one method or layer of your implementation into a test case, but you should not cram every feature onto only a couple test cases to save time. Cutting corners like this is just asking for defects to be released, for code to not be fully tested in the proper way, or just an overall low quality product. Writing thorough tests and maintaining a nice organized test bench is invaluable as a project scales up, and having proper and scalable code in the implementation makes everything more efficient in the future. If technical debt is not addressed, it can accumulate over time and become more difficult and expensive to fix. As a practitioner in the field, it is essential to plan and prioritize the management of technical debt to prevent it from becoming overwhelming. This can be achieved by consistently refactoring and improving code quality, prioritizing testing, and ensuring that any new code is written with scalability and maintainability in mind. For example, imagine a software engineer working on a new feature for an e-commerce website. They are under pressure to meet a tight deadline and decide to cut corners in the testing phase to save time. They release the feature to users, only to discover that it contains a significant bug that causes the website to crash frequently. This leads to a loss of user trust and revenue for the company, as well as additional resources and time spent fixing the issue.