Zach Fizet

CS-320

4/15/2025

Project Two Summary

My unit testing approach for each of the three features closely followed with that of the rubrics that were given to us for each different milestone. As part of my methodology for creating code that was true to the requirements, I ensured that I was creating features that were specifically asked for without adding any unnecessary things that weren’t directly referenced as a direct requirement. For example: In the Contact Service milestone, we were asked to create two classes (Contact.java and ContactService.java). In the rubric, we were given distinct requirements for each class. The Contact class needed to create Contact objects that held certain data that would be used by the Contact Service class. These were things like contact IDs, first names, last names, phone numbers, and addresses for each contact. For each of the milestones, I started first by creating the class that was going to be utilized by the other “service” class. I did this because in my mind, it was easier to first create the objects we were going to be working with before manipulating them later in the service classes. By creating the object classes first, I was ensuring that all of my requirements were met for both classes because the requirements would often overlap, meaning that one could not exist without the other. Referencing the Contact classes again, one of the requirements of the Contact Service class was that we would be able to update the parameters established by the Contact class. If my Contact class didn’t have a way to store and access one of those parameters (like a last name), my method updateLastName in my Contact Service class wouldn’t be able to function properly because I was missing a requirement from the Contact class. This type of methodology was something that I would follow throughout the other milestones as we continued to create CRUD (create, read, update, delete) operations throughout.

When creating my code, I ensured that it was technically sound by utilizing specific techniques. One of the techniques that I used in my code was unit testing with Assertions. Assertions confirm that a specific method is behaving properly. For example, in my TestServiceTest class, I have this line of code: assertThrows(IllegalArgumentException.class, () -> service.addTask(task2));. This line of code will ensure that adding a task with a duplicate ID is correctly blocked by the program. This technique of Assertion testing ensures that I am maintaining data integrity by making sure the program doesn’t allow for the use of duplicate IDs. Another technique that I used to ensure my code was technically sound was the use of input validation. Input validation protects the code from bad or invalid inputs before they cause any major errors or crashes within the program. For example, in my Appointment class, the constructor utilizes this line of code: if (date == null || date.before(new Date())) {

throw new IllegalArgumentException("Invalid appointment date");

}. This checks the date of a new appointment that is being created and prevents it from being created if that date is in the past. Exception handling is also helpful for making issues easy to identify by using descriptive language like: “Error task ID already exists” or etc. That way you can easily see why that error is being generated. To ensure my code was efficient, I utilized HashMaps for fast lookup options. Almost all of the features implemented involved updating or creating specific pieces of data that were tied to specific, unique objects. HashMaps are great for this because they utilize keys like task ID’s or contact ID’s to find a specific object. This is helpful for ensuring that your code is efficient, because if we were to utilize another data structure, we might have to iterate throughout the entire list or tree full of different objects before we got to the one we were looking for. If we’re working with large amounts of data, this can greatly increase the runtime of the program. By using a HashMap, it makes search, update, or delete functions fast regardless of how much data we’re working with.

The main software testing technique that I utilized throughout this entire project was JUnit testing. JUnit testing is when each class and method is tested in isolation using JUnit. JUnit tests are automated test cases that are annotated with @Test that contains assertions to check if the output of the specific method matches the behavior that was anticipated. For example, in the TaskTesk class, there is a method called testTaskCreation that I utilized to verify that the constructor sets the values correct. Using assertEquals compares the expected vs. actual values. Another JUnit technique that I utilized in this project was testing for invalid object creations by using assertThrows. assertThrows will product an exception when it is triggered by an invalid input. For example, in my TaskTest class, I test for invalid task creation by using assertThrows to ensure that an exception is being thrown when a task is created with a “null” task ID. The requirements of the Task class required that task IDs not be null, so this specific test was utilized to ensure that an exception would be thrown if a task was created with a null task ID.

During my time developing this project, there were a few different kinds of testing that I never made use of. One of those testing methods is Integration Testing. Integration Testing tests how multiple components of a program interact together and with something like a database. For example, let’s say a database existed already with a company that had a large number of contacts already saved and generated from a program they were using in the past. Integration testing would test how the program I developed would interact in handling those contacts that already exist. If they didn’t have the same parameters that I have established within my Contact class, it probably wouldn’t work very well, and I would need to develop some sort of class or method that would take the contact information within the database and convert it into objects that work with my program that I have created. Another form of testing I didn’t use during my time developing this program was Regression Testing. Regression Testing is utilized when updating code to ensure that the rest of your program still functions after the update. This would be applicable if a bug or something were to be found in your program and you needed to update the code to fix it. Part of fixing that bug would involve regression testing, ensuring you didn’t break anything else when fixing whatever code wasn’t functioning properly. I didn’t use this testing because we didn’t really flesh out a full, functioning application that was deployed for use. JUnit testing was sufficient for what we were tasked with completing during the development of this project.

During my time developing this code, I didn’t really employ a “ton” of caution, but I wanted to ensure I wasn’t developing tests that were unnecessarily complex. My main areas of “caution” I guess would be just things that I consider to be “standard coding practices”, things like testing invalid data and edge cases. These tests are important because of how they integrate into backend validation. I took a software security class a few terms ago, and one of the topics we covered had to do with “malicious inputs”, where critical data was accessed through the use of manipulating inputs in various fields like usernames and passwords. I believe the specific case we learned about was where an unauthorized user could access any account on a website as long as they entered the full URL into one of the input fields. This would provide unwanted access to potentially sensitive information to users that shouldn’t have access to that information. In my code, I use specific constraints to limit invalid inputs. This can be seen throughout different methods in my code, but as an example: if (name == null || name.length() > 20) {

throw new IllegalArgumentException("Invalid name");, this line of code follows the requirements we were given where we were asked to limit the maximum length of a name. This could be problematic if you have a longer name, but will generally fit most people. Additionally, I placed constraints on other things like task names, contact phone numbers, and more. These limitations are important because they maintain the integrity of the data being input into the system. These constraints are coded to check and catch for input errors are the earliest point (during object creation), preventing any invalid data from entering the system and potentially causing major issues further down the line. I also use proactive defensive programming throughout my code by utilizing lines of code like: if (taskMap.containsKey(task.getId())) {

throw new IllegalArgumentException("Task ID already exists.");. These methods are also tested using JUnit tests that confirm exceptions are thrown when they are programmed to be thrown. Lastly, my exceptions are also accompanied by clear error messages which help for debugging and reduces the mental overhead if someone were to maintain or use my code later on.

During my time developing my code, I avoided bias by trying to shift my mindset to that of an external tester. I developed my tests after developing my methods and I included negative test cases to ensure that things were “correctly” failing, not just working successfully. As an external tester, it would be helpful to see that tests that were coded to fail were operating as such, because an external tester won’t just be testing your code to ensure it’s functioning properly, but also failing properly when it’s supposed to. Writing these failures and validation logic into my code helps testers in the future as well by reducing unnecessary debugging time, therefor saving development time overall by “frontloading” the development of tests as I was developing the project.

Discipline and quality matter when developing code because code is read more often than it is written. When you develop a project, dozens of people will come after you and read your code, you might move onto another company while the previous company you worked for hires new people that will read code you wrote years ago. The code you write isn’t just solving a problem in the present, but you’re essentially leaving instructions for future developers, sometimes even yourself. If the code you write isn’t clean, well-structured, or formatted well, you’re creating future problems for yourself and others. The readability of code is crucial to the future success of you and your development team. If someone can’t look at your code and easily understand it, it will only create confusion in the future. In my code, I ensured readability by formatting it as cleanly as possible and leaving robust comments where necessary. Comments are important because even if your code looks messy, a detailed comment will at least let future readers know what that block of code is supposed to do. By following best coding practices like this, you’re preventing yourself from incurring a technical debt for the future. Another way to prevent yourself from incurring technical debt, (and potentially financial debt) in the future is by testing. Catching a bug as early as possible will always be cheaper than catching a bug later on in the software development process. It also becomes much harder to debug a problem you should’ve caught earlier when your program has progressed far beyond that problem. For example, if my Appointment class didn’t validate date inputs, and a user sets a meeting to the year 1900, if this date were to be imported into a calendar program or something else within a potentially intricate system, it could crash the application or ruin its functionality. My JUnit test that checks assertThrows for past dates catches this issue as early as possible, preventing it from turning into a much larger problem in the future. This is an example of my quality discipline paying off. My father was a carpenter for many years of his life, and growing up he always told me “Measure twice, cut once”. This is a principal I have carried with me throughout my entire life, and I will continue to abide by that principal as a guide for the rest of my life, both professionally and personally.