Corey Gaspar

10/15/2025

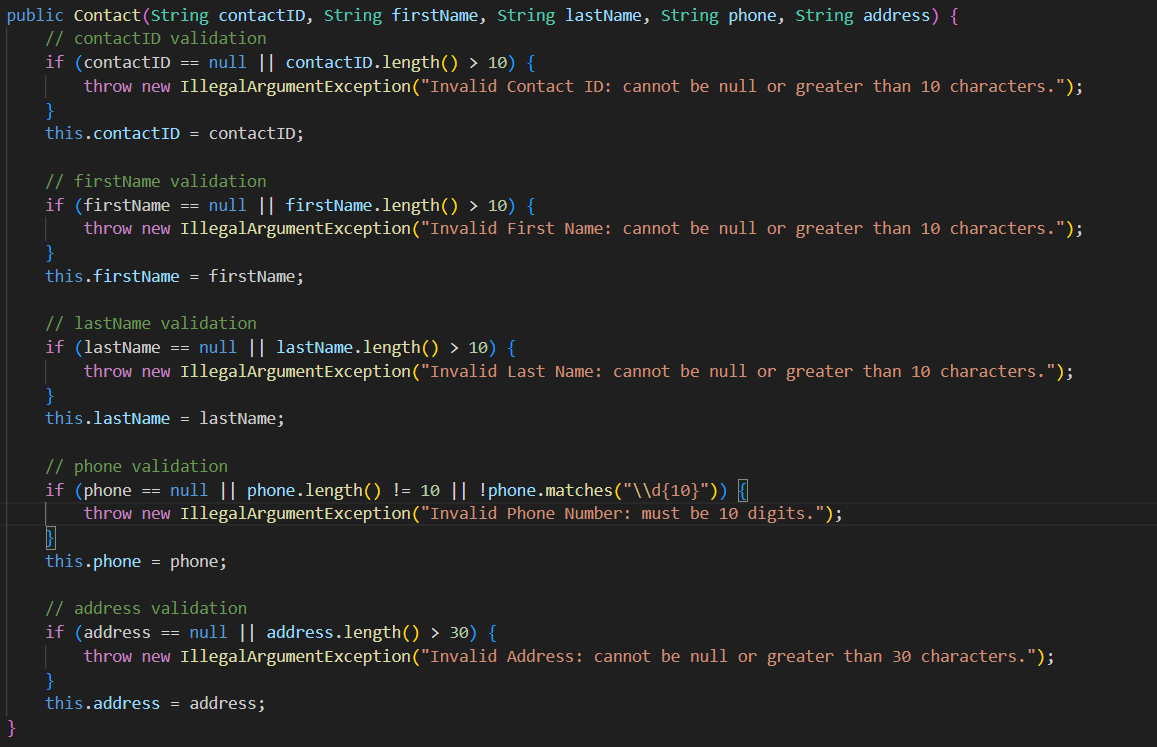
**Project Two**

**Summary**

**1.1 Alignment to Software Requirements**

My unit testing approach directly aligned with the software requirements because each milestone laid out the requirements for each service and what “rules” the service is expected to follow. In Module 3, I was asked to create the contact service of the program. The service included a Contact class, which declared all of the fields defined by the requirements, along with validation for those fields. According to the requirements, the contact ID could not be null, updated, or longer than 10 characters. The first and last names could not be null or exceed 10 characters, the phone number could not be null and had to be exactly 10 digits, and the address could not be null or longer than 30 characters.

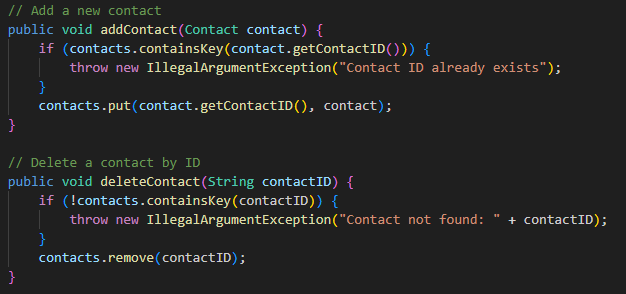
To ensure that the contact ID could not be changed, I set the contactID string to “final”. To verify that my fields were meeting the milestone requirements, I implemented input validation checks in the constructor as shown in the code snippet below:

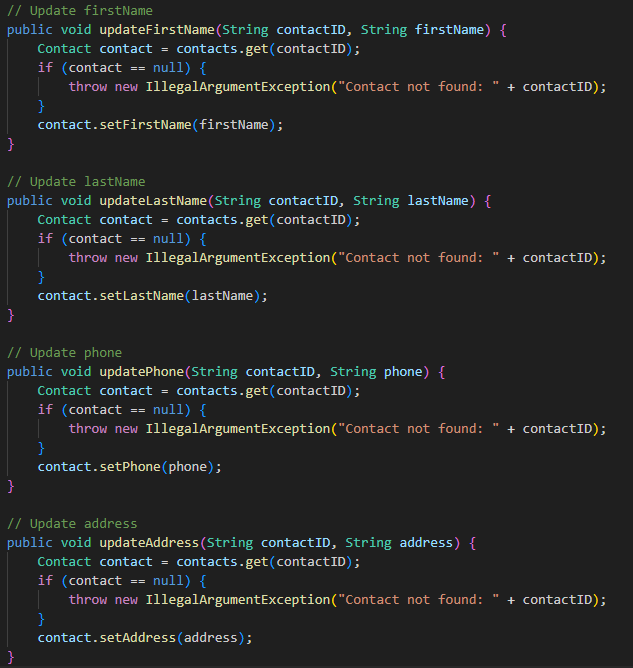
****

This ensured that any input entered by a user that does not meet these requirements would be rejected. Not only that, but it ensured that my code meets the requirements of the service. The same logic was applied to the Appointment and Task classes to ensure that I was meeting the requirements of the milestones.

Once the Contact class was complete, I moved onto the contact service, ensuring that it met the functional requirements outlined in the instructions of the milestone. The service needed to be able to add contacts with a unique ID, delete contacts using the contact ID, and update the first name, last name, phone number, and address fields. To meet these requirements, I decided to use a HashMap to store the contact objects in memory, using the contact ID as the key to satisfy the uniqueness requirement. Before adding a contact, the service checked whether the ID already existed, throwing an exception if it did exist. The delete and update methods also used the contact ID to find contacts and modify them, with each update method called the correct setter from the Contact class. This can all be seen in the code snippets below:



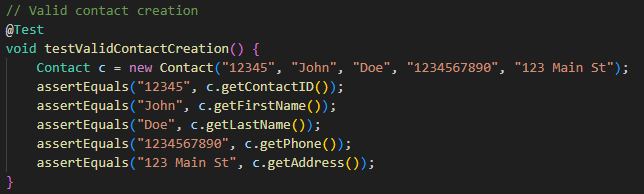




The Appointment and Task services followed the same logic, using a HashMap and unique IDs to ensure all requirements were met. This approach ensured that my code and testing fully aligned with the software requirements for each milestone.

**1.2 JUnit Test Quality**

The quality of my JUnit tests was high because they fully verified that all functional and validation requirements were met for both my classes and services. For my Contact class, my tests confirmed that valid contacts could be created successfully, while invalid contacts would throw invalid exceptions.

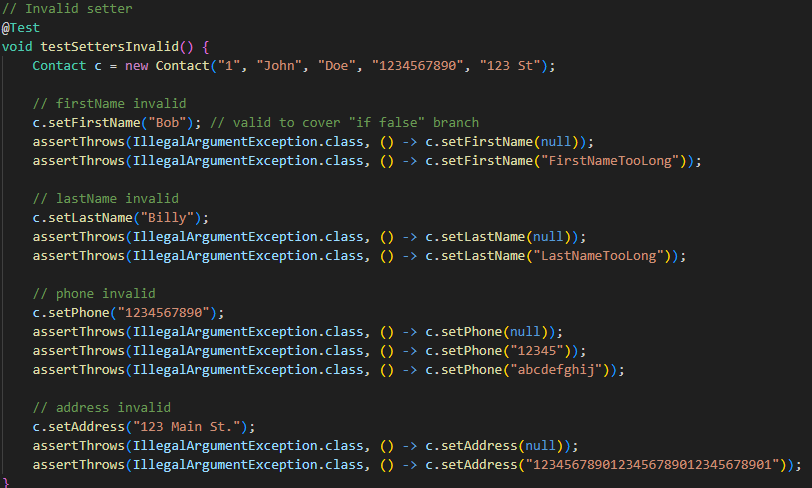




In addition to testing valid and invalid constructors, I also verified that boundary conditions for each field were being handled properly, including minimum and maximum lengths for IDs, names, phone numbers, and addresses. I also tested my setters to ensure that valid updates were being applied and invalid updates were being rejected. The same logic for my JUnit tests was applied to the other classes as well.







For the contact service, my tests verified that contacts could be added with unique IDs, deleted by ID, and updated correctly for the first and last name, phone number, and address. I also ensured that attempting to create duplicate contacts or modify non-existing contacts would throw exceptions.

After running my JUnit tests, I reviewed the test coverage and noticed that all of my classes and services reached 100% coverage. This means that every line of code in those files was executed during testing, including normal operations and other edge cases.

**Reflection**

**2.1 Software Testing Techniques**

The main software testing technique that was employed for this project was unit testing using JUnit. Unit testing focuses on testing individual pieces of a program, like classes and methods, to make sure they work well on their own before being implemented into a larger system. Throughout each week, I tested my classes and services individually to ensure that they met all functional requirements and handled all invalid input correctly. I also used boundary testing within my unit tests to check the limits of each field, such as making sure IDs, names, phone numbers, addresses, and descriptions did not exceed their max lengths. Finally, I utilized integration testing to ensure that each piece of the program worked correctly when combined together into one system. Using these techniques helped me test as many possible edge cases I could, which led to my 100% test coverage.

**2.2 Additional Testing Techniques**

Some other software testing techniques that I didn’t use in this project include system testing, regression testing, and acceptance testing. System testing focuses on verifying that entire program works as a complete system, ensuring that all pieces work as intended together. Regression testing is done when pushing changes and updates to the software to ensure that the existing features of the program still function properly. Finally, acceptance testing is typically used in real-life scenarios when dealing with end users or clients. This type of testing is done by the end user or client to ensure that the software meets their needs.

These techniques were not used because this project mainly focused on developing and testing the individual classes and services, rather than creating a full software system. Since the project was limited to the development and testing of these classes and services, unit testing and integration testing were sufficient enough to ensure that they met requirements. If I were to take what I built and turn it into a full software system, then those more advanced testing techniques would need to be used to confirm that it functions as intended. The testing techniques listed here are better suited for larger scale projects.

**2.3 Practical Uses and Implications**

Each of the testing techniques mentioned have specific uses and benefits depending on the type and size of the software project. Unit testing is most effective during the early development stages, especially for smaller projects, because it helps developers identify bugs and implement fixes before integration. Unit testing helps save developers time and money because it prevents larger issues from occurring later on. Boundary testing is especially useful when developing software that works with user input or data validation, since it ensures that software properly handles values at or near the limit of allowed input. For example, if you were to have an input field where a user could enter their first name and you only want it to be 10 characters, then you can use boundary testing to make sure that if it is over 10 characters, an error will be thrown. Integration testing is beneficial to making sure that each “unit” works correctly together, which is important if you are working with an application that relies on multiple components, especially in a larger-scaled system.

For larger, more complex projects, system testing confirms that each individual unit functions as a complete system and meets performance and reliability expectations. Regression testing is most useful for projects that are being updated regularly, ensuring that new features and changes do not break the preexisting functionality. Finally, acceptance testing plays an important role when developing client-based or commercial software because it allows the end user to verify that the final product meets their requirements and needs.

**Mindset**

**3.1 Employing Caution**

While working on the milestone and project, I was careful and thoughtful when developing the code. I first thought like a security tester, considering all possible ways that the program could fail or be used incorrectly. I tested all inputs, including valid and invalid values, to make sure the program handled them correctly. This was important because even the smallest mistake, like duplicate IDs, could cause issues within another part of the program. For example, in ContactService, I checked that adding a contact with an existing ID caused an error and that updates only affected existing contacts. Considering all possible edge cases helped me create reliable and complete tests.

Other than thinking like a security tester, I also thought like the client. I considered how a user would interact with the program and what problems they might encounter. For example, I checked that adding, deleting, or updating a contact worked as expected and gave clear errors when something went wrong. This helped me make sure the program was easy to use, reliable, and met the requirements from both a developer’s and user’s perspective.

**3.2 Limiting Bias**

Throught this project, I tried to limit bias by testing every part carefully, even parts I thought were correct. I made sure to check both valid and invalid inputs and included boundary cases to see how the program handled unusual situations. For example, in ContactService, I tested adding a contact with a duplicate ID, even though the code I wrote should prevent it, to confirm it actually worked as expected.

As a software developer, bias can be a concern when testing your own code because it’s easy to assume your code works correctly and overlook mistakes. For instance, if I only tested adding contacts with valid, unique IDs, I could have missed that the program could allow duplicates or fail with invalid phone numbers. By actively searching for errors within my code, even if I thought there were none, I was able to limit bias and ensure that my code ran correctly.

**3.3 Staying Disciplined**

Being disciplined is very important in software engineering because it ensures that code is reliable, maintainable, and meets the required standards. Cutting corners when writing or testing code can lead to bugs, security issues, or failures later on, which can be expensive and time-consuming to fix. For example, if I had skipped validation checks in the Contact class, the program could allow invalid phone numbers or duplicate IDs, causing problems in other parts of the system.

To avoid technical debt, I would follow industry best practices such as writing clear, well-documented code, carefully performing unit and integration testing, and reviewing and refactoring code when needed. By consistently testing for edge cases, validating inputs, and ensuring my services handle all expected scenarios, I can prevent problems from occurring. Problems are inevitable, but staying disciplined allows me to produce high-quality software that is simple to maintain and fix later on.