The journey of building and running code transforms into a craft, shaped by the lesson of failure. I was new to Junit, a popular framework for writing unit tests in Java. Reading through a few assignments, I learned how to evaluate methods, handle edge cases, and ensure that my code was dependable and maintainable. This experience taught me how to improve my debugging skills and overall shape the way I approach problems and the requirements that follow behind. In the beginning of each section, I started off organizing the requirements:  
 **Contact Class Requirements**

* Contact ID: must be unique, not null, no ≤ 10 characters, and cannot by changed once set.
* First Name: required, not null, and max 10 characters.
* Last Name: required, not null, and ≤ 10 digits.
* Phone Number: required, not null, and = 10 digits
* Address: required, not null, and must be no longer than 30 characters.

In accordance with the Contact Service requirements, I needed to ensure that new Contact objects could only be added if their contact ID were unique. The system prevents duplication by validating that no existing contact shares the same ID. Additionally, the service must support removing a contact by providing its specific contact ID, ensuring targeted and reliable deletion.

**Contact Service Requirements**

* Add Contacts with Unique ID: the service must allow new Contacts objects to be added only if the contact ID is unique.
* Delete Contacts by Contact ID: should be able to contact by supplying its unique ID

contactList.remove(contactId);

* Update Contacts Fields by Contact ID: ensure the fields can be updated (firstName, lastName, phoneNumber, address)

**Task Service Requirements**

* Add Tasks with Unique Task IDs: service must be able to add new Task objects as long as each has a unique ID. Map<String, Task>
* Delete Tasks by Task ID: must be able to remove a task using its unique identifier.
* Update Task Fields: can update certain fields of a task, such as taskName, description, dueDate. Like with contacts, task ID should not be editable once created.

**Task Class Requirements**

* **Unique Task ID**: must be null, not exceed 10 characters andcannot be updated after creation.
* **Name Field**: required (not null) and max length is 20 characters.
* **Description Field**: required (not null) and max length is 50 characters.

**Appointment Class Requirements**

* **Appointment ID**: must be unique, not null, cannot exceed 10 characters and updateable after creation.
* **Appointment Date**: required (not null), follow YYYY-MM-DD format and should not be in the past.
* **Description**: required (not null), cannot exceed 50 characters, and should be brief and relevant to the appointment.

**Appointment Class Requirements**

* **Add Appointments with a Unique Appointment ID**: service must allow adding new Appointments objects, ensure no duplication and each appointment must have a unique ID that does not already exist in the system.
* **Delete Appointments by Appointment ID**: service must support deleting a specific appointment using its unique ID.

The overall quality of my JUnit testing was based on code coverage and validation. I want to make sure that I implemented the correct code and execution of their requirements. I achieved a good 80% coverage in the classes under test. My execution of the code seemed unsuccessful the first 4-5 times but after achieving a slow and step by step to include all conditional branches such as if-else blocks, exceptional in managing the paths and using critical thinking in logical paths. I incorporated tests that intentionally targeted that typically use cases, edge cases, and negative as well. Covering the right parts of the code with the right assertions and the ensure metrics back it up. Writing Junit tests was a process of trial, insight, and continuous improvement. Each bug or syntax error increased my confidence in the power of not giving up on the development. In the following example, you will see the code contact IDs were properly validated in my Contact class:

@Test

public void testContactIdTooLongThrowsException() {

Exception exception = assertThrows(IllegalArgumentException.class, () ->

new Contact("01234567890", "Alice", "Smith", "1234567890", "123 Main Street"));

assertEquals("Invalid contact ID", exception.getMessage());

}

I structured my tests to avoid repetitive setup logic, using the @BeforeEach annotation to initialize shared resources once before each test.

@BeforeEach

public void setup() {

contactService = new ContactService();

sampleContact = new Contact("0001", "Bob", "Lee", "1234567890", "742 Evergreen Terrace");

contactService.addContact(sampleContact);

}

Efficient assertion logic (comparing expected and actual outcomes) helped my isolate failures quickly.

In further of my JUnit reflection, unit testing focused on testing individual methods/ components in isolation to confirm they perform as what’s expected. Outlining a fast execution, easy to automate, detection of bugs early, and help enforce modular code design. I personally felt that I should have implemented more system testing prior to deployment because it is how I verify the results of a product- ensuring all functional and performance requirements are met. In the beginning, development of the code was an obstacle because of the simpler errors, missing components and losing sight of what I expected of the output. For practical use and implications, Unit testing is ideal during the development phase when you are actively building or refactoring code. Integration testing becomes critical in multi-class projects, especially with databases. Exposing issues that unit cannot detect.

In the end, I needed to adopt a tester mindset. I learned to approach testing with a cautious and critical mindset. Rather than assume my code would work, I worked under the belief that bugs were hiding, and my part was to expose and clean the code. For example:

new Contact("123", null, "Smith", "1234567890", "123 Main Street");

This helped safeguard proper handling or null values and enforced required constraints.

The importance of understanding parts of the code is essential. Recognizing that slight changes in one area could cause a water ripple of failures somewhere in the code. When reviewing my own code, I understand that bias can easily creep in when evaluating your code- especially when you subconsciously avoid evaluating your scenarios where you can predict it will fail. In my own words, I created slow step creation to break my assumptions or unhealthy habits. Simple creation to code requires discipline and this determines the quality of code. Skipping over unit tests, not delaying error handling, and not hard-coding values for now. I want to avoid future changes and risker moves, combating using refactored verbose logic into smaller, testable components or even write reusable utility methods in tests to prevent duplication. In the long term, I plan to perform regular refactoring and treat testing as an integral part of planning out code. Avoiding technical debt is not about being perfect but executing a consistent and initiative-taking approach.

Sources

[Module Five Milestone Guidelines and Rubric - CS-320-10630-M01 Software Test, Automation QA 2025 C-3 (May - Jun)](https://learn.snhu.edu/d2l/le/content/1918304/viewContent/40479897/View)

[Module Three Milestone Guidelines and Rubric - CS-320-10630-M01 Software Test, Automation QA 2025 C-3 (May - Jun)](https://learn.snhu.edu/d2l/le/content/1918304/viewContent/40479894/View)

[Module Four Milestone Guidelines and Rubric - CS-320-10630-M01 Software Test, Automation QA 2025 C-3 (May - Jun)](https://learn.snhu.edu/d2l/le/content/1918304/viewContent/40479895/View)

[CS 320 Module Four Tutorial Using JUnit](https://learn.snhu.edu/content/enforced/1918304-CS-320-10630.202551-1/course_documents/CS%20320%20Module%20Four%20Tutorial%20Using%20JUnit.pdf?ou=1918304)