Project Two: Summary and Reflection Report

Cody Vachon

Southern New Hampshire University

CS 320: Software Testing, Automation, and Quality Assurance

Instructor: Omar Toledo

April 26, 2025

1. **Summary**

This report summarizes the unit testing approach used for the Contact, Task, and appointment services implemented for an application. It continues to evaluate the alignment of the testing approach with requirements and defends the overall quality of these tests and the writing experience.

**Unit Testing Approach**

Unit Testing Approach and Alignment with Software Requirements

**Contact Service:**

Unit testing approach focused on verifying the behavior of the Contact class and ContactService class in isolation.

The following are functionality requirements for the contact class:

* A contact object is created successfully with valid input.
* Exceptions are thrown for invalid input.

The following are functionality requirements for the contactservice class:

* Able to add a new contact to the service.
* Throwing an exception when a contact is added to an existing ID.
* Deleting a contact from service.
* Throwing an exception when updating a non-existent contact.
* Ensure null values that update fields do not modify the existing data.

Evidence of alignment for Contact Service:

This action was validated by first creating a well-formed contact object, uniquely identified by "12345", and then calling the addContact method. Subsequent verification confirmed the successful storage of this contact within the service's internal collection, accessible via its unique ID.

Evidence: The assertion assertEquals(contact, service.contacts.get("12345")); definitively demonstrates that the newly added contact is stored and retrievable using the designated key, "12345". This outcome directly fulfills the functional requirement for the service to accurately store and retrieve valid contact information based on a unique identifier.

A screenshot of a computer

AI-generated content may be incorrect.

**Task Service:**

The unit testing for task service approached testing the task class.

The following are functionality requirements for the task class:

* Successful creation of a task with valid input.
* Exceptions are thrown for invalid input.

The following are functionality requirements for the taskservice class:

* Able to add a new task.
* Throwing an exception when a task with duplicated ID.
* Deleting a task.
* Throwing an exception when updating a non-existent task.
* Retrieving task by ID.

Evidence of alignment for Task Service:

A key objective of the TaskService is to ensure the uniqueness of task IDs. This test explicitly addresses this by first successfully adding a task and then attempting to add another with the same identifier. The expected IllegalArgumentException upon the second attempt confirms the service's enforcement of ID uniqueness.

Evidence: The test procedure involved adding a task named "task1" and subsequently trying to add "task2" with the identical ID "1234567890". The successful execution of assertThrows(IllegalArgumentException.class, () -> service.addTask(task2)); provides direct evidence that the service correctly identifies and rejects attempts to create tasks with non-unique IDs, aligning with the stated functional requirement.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Appointment Service:**

The unit testing for the appointment service included testing the appointment class and appointmentservice class.

The following are functionality requirements for the appointment class:

* Successful creation of a appointment with valid input.
* Exceptions are thrown for invalid input.

The following are functionality requirements for the appointmentservice class:

* Able to add a new appointment.
* Throwing an exception when an appointment is added with null or duplicate ID.
* Deleting an appointment by its ID.
* Throwing an exception when an appointment is deleted with a non-existent or null ID.
* Retrieving an appointment by its ID.

Evidence of alignment for Appointment Service:

A critical aspect of robust system design involves the proper handling of invalid inputs. This test specifically targeted the requirement that the system must reject null appointment objects to maintain data integrity and prevent unexpected behavior. The successful triggering of an IllegalArgumentException demonstrates the service's adherence to this crucial input constraint.

Evidence: The direct invocation of service.addAppointment(null) intentionally provides an invalid input to the system. The subsequent assertion that an IllegalArgumentException is thrown unequivocally confirms that the service effectively identifies and rejects null appointment values. This behavior directly validates and reinforces the software requirement that the system must not accept null appointments, thereby safeguarding system stability and data integrity.

A screen shot of a computer

AI-generated content may be incorrect.

Overall Quality of Junit Tests and Coverage Percentage

The quality of JUnit tests and the project’s coverage improve when there’s a focus on a few key areas: good coverage, isolation, readability, and following best practices. With comprehensive coverage, all the critical paths, including those tricky edge cases and negative scenarios, get tested properly, leaving almost no gaps in functionality checks. Tests are set up to be isolated too, concentrating on just one unit of functionality without any external dependencies, which helps in getting accurate results.

Readability is also important—clear method names and a logical test structure make it easier for future maintainers to follow what's happening. Following best practices like meaningful assertions, modular design, and consistent error handling makes the tests effective and maintainable. When all these elements come together, they ensure thorough verification of the code and support high-quality software development.

**Experience Writing the Junit Tests**

Experience Writing Junit Tests and Technical Soundness

Example from ContactTest.java:

The technical soundness of the code in this test is ensured through the following approaches:

* Valid Input Check: The test uses realistic input for the Contact constructor, ensuring it works correctly with normal data.
* Field Checks: Each part of the Contact object—contactID, firstName, lastName, phone, and address—is verified, confirming the constructor sets everything properly.
* Thorough Testing: The test checks all fields to ensure no important details are missed during object creation.

These practices ensure that the constructor functions correctly during object creation.

A screen shot of a computer

AI-generated content may be incorrect.

Example from TaskServiceTest.java:

The technical soundness of this test is ensured through the following approaches:

* Focused Scope: The test solely examines task name updates, ensuring a clear focus on the specific logic.
* Isolation: By using new instances of TaskService and Task, the test avoids external states, preventing interference from other tests.
* Edge Case Consideration: The test targets updating a task by ID, covering a typical use case to ensure reliability.

These practices confirm that the tested code is technically sound and behaves as intended.

A screenshot of a computer program

AI-generated content may be incorrect.

Example from AppointmentServiceTest.java:

The technical soundness of the provided code is ensured through the following methods:

* Realistic Setup: The test initializes an AppointmentService and uses a valid Appointment object with a future date, simulating real-world scenarios for proper data handling.
* Core Behavior Validation: It verifies that the addAppointment and getAppointment methods function correctly by checking that the retrieved appointment matches the one initially added.
* Precise Targeting: The test focuses on retrieving an appointment using a valid ID, keeping the scope narrow for accurate troubleshooting.

These practices confirm that the method behaves as intended, handles time-sensitive logic appropriately, and maintains accuracy in storing and retrieving appointments.

A screenshot of a computer code

AI-generated content may be incorrect.

Efficiency of JUnit Tests

This project achieves JUnit test efficiency through four main methods: fast execution, isolation, targeted testing, and parallel execution. Fast execution comes from using lightweight tests that focus on single functions, which reduces extra work and allows for quick feedback during development. Isolation involves testing each component separately to prevent issues from other parts of the system and cut down on setup time. Targeted testing focuses on specific functions, reducing unnecessary repetition and speeding up execution. Lastly, parallel execution allows multiple tests to run simultaneously, greatly increasing efficiency and scalability, especially in larger codebases. Overall, these practices help ensure that the project functions are validated quickly and reliably.

ContactServiceTest.java:

The use of efficient methods, where this test did not involve any external dependencies. Where the class operated solely on in-memory data structures, illustrated below:

* Fast Execution: The test quickly verifies the addition of a valid contact to the ContactService with minimal setup, ensuring rapid execution and immediate feedback.
* Isolation: It is self-contained, focusing solely on adding a contact by creating new instances of ContactService and Contact, preventing interference from other tests.
* Targeted Testing: The test specifically checks the addContact functionality using assertEquals for precise validation, keeping the focus clear.
* Parallel Execution: It is designed to support parallel execution since it does not share state or data with other tests, allowing efficient execution.

By adhering to these principles, the test ensures efficient validation of the addContact functionality, contributing to the overall speed and reliability of the project's test suite.

A screenshot of a computer

AI-generated content may be incorrect.

TaskServiceTest.java:

The use of efficiency was similarly in the task service, operating on memory. The TaskServiceTest.java is illustrated below:

* Fast Execution: The test focuses on adding a valid task and verifying the result without unnecessary setup, ensuring quick execution and immediate feedback.
* Isolation: It operates independently by creating new instances of TaskService and Task, preventing interference from other tests and avoiding reliance on shared state.
* Targeted Testing: The test specifically validates task addition by asserting that the task can be retrieved using its ID (assertEquals(task, service.getTask("1234567890"))), keeping the focus clear.
* Parallel Execution: The self-contained nature of the test allows it to run efficiently alongside other tests without altering global state.

These principles contribute to a fast, reliable, and efficient validation of the task addition functionality in TaskService.

A screenshot of a computer code

AI-generated content may be incorrect.

1. **Reflection**

This project was created from three milestone projects, each used to test different areas of a data server that models a user’s ID to access a contact, task, and appointment. These modules were implemented to perform tests throughout the project. The testing technique used in the contact model was unit testing. The task module incorporated integration testing. The appointment module used system testing.

**Testing Techniques**

Testing Techniques Employed

The importance of unit testing in software development. I learned that unit testing helps catch bugs early by testing individual components of the code in isolation. We explored how to implement these tests in our contact service project, using tools like JUnit and Mockito. It was fascinating to see how writing specific test cases for different scenarios can significantly improve the reliability of our code. Overall, it gave me a deeper appreciation for the role of testing in creating high-quality software.

Then moving on to the task service project focused on integration testing. This project focused on how the other individual components work together with other codes. We explored how to implement these tests in our task service project.

Finally, we used automated testing for system testing. This testing was used on appointment service for the flow of the application performance on the environment. The project focused on the booking appointments, receiving confirmations, with other flow.

Other Software Testing Techniques Not Used

The performance testing wasn’t used during this project. It's not just about whether the app works, but how well it works under pressure. Imagin thousands of users simultaneously trying to access the database, and the whole application crashing’s due to the burden on the servers. The performance test would need to implement later with the application before deployment.

The user acceptance testing wasn’t employed. The projects focused on getting the code to compile and pass our internal tests that we haven't really considered how actual users will interact with the final product. We're making assumptions about what they wanted. The last thing we need is to rebuild major chunks of the application post-launch.

Practical Uses and Implications of Testing Techniques

Software quality is ensured through a tiered testing approach. Unit testing focuses on verifying the correctness of individual code components in isolation, enabling early bug detection and providing a safety net for refactoring, particularly crucial for reusable libraries or complex logic. Integration testing then validates the interactions between different modules or services, ensuring seamless data flow and interface compatibility, which is vital for microservices architectures and systems with external dependencies.

Finally, system testing evaluates the entire application flow in a realistic environment, confirming end-to-end functionality, performance, and overall user experience, especially critical for customer-facing applications and complex business processes. This multi-layered strategy, from individual units to the integrated system, contributes significantly to the reliability and stability of software development projects.

**Mindset**

Mindset: Caution, Complexity, and Interrelationships

The mindset that was used throughout this project can be characterized with disciplined thoroughness, that was attentive to detail, and had proactive approach to identify potential issues. The code was questioned to assume any defect that may exist.

Caution:

Caution was key in this approach. For example, in the test named `contactConstructor\_InvalidPhone\_ThrowsException`, the tester looked at all kinds of invalid phone number scenarios. They checked for things like numbers being too short or too long, null values, and non-numeric entries. This level of caution is really important because it helps prevent runtime errors and strengthens the application by tackling issues before they escalate.

Example from ContactTest.java:

A black and green screen with text

AI-generated content may be incorrect.

Complexity and interrelationships:

Understanding code complexity was also substantial. The tester could identify potential failure points by recognizing how the Contact constructor interacts with other system components. Validating inputs upfront ensures that downstream processes receive only correct data, avoiding bigger problems later.

Finally, acknowledging the connections between components was vital. Thoroughly testing the constructor prevents bad data from entering the system, protecting other features from potential failures.

Mindset: Limiting Bias

An effort was made limit bias in the code by using a systematic approach in reviewing this software. Predefined test cases and acceptance criteria was prioritized.

The following techniques were used:

* Test Case Documentation
* Code Reviews
* Focus on Requirements
* Automated Testing

Bias in Developer Testing:

Limiting bias in testing requires a mindset that actively seeks flaws and scenarios beyond the developer’s assumptions. Objective processes like systematic edge case testing and peer reviews ensure the application is resilient and behaves as expected even in unexpected situations.

Examples:

Specific examples, such as the contactConstructor\_InvalidPhone\_ThrowsException, highlight the importance of rigorous testing for edge cases, invalid inputs, and unanticipated scenarios—key steps to building robust and error-free software.

@Test  
void contactConstructor\_ValidInput\_WorksCorrectly() {  
 Contact contact = new Contact("12345", "Jane", "Doe", "1234567890", "456 Elm St");  
 assertEquals("1234567890", contact.getPhone());  
}

This test validates that the Contact object is created with valid inputs, but it entirely skips invalid input cases:

* **Null phone number** (null)
* **Non-numeric characters** ("abc123xyz")
* **Empty string** ("")
* **Boundary length failures** ("123" or "123456789012345")

As a result, exceptions for invalid inputs aren’t tested. When such inputs are encountered in production, they may cause unpredictable errors—errors that could have been avoided with unbiased testing.

Mindset: Discipline and Commitment to Quality

Discipline and commitment to quality are crucial for software engineers, as it ensures the creation of functional and reliable software. Prioritizing quality helps avoid immediate problems and supports long-term success for teams, clients, and users. This dedication means delivering systems that work well under various conditions and preventing issues that could hinder growth.

Why it’s important not to cut corners

Not cutting corners in coding and testing is key to preventing future failures. Skipping edge cases or validation can let bugs slip through, leading to significant issues down the line that affect user experience and increase fixing costs. For example, if we hadn’t tested the Contact constructor for invalid phone numbers, users could create contacts with null or bad data, causing crashes in other services.

Reliable, well-tested code builds trust among clients and users. Basic errors from overlooked steps can quickly damage credibility. The test `contactConstructor\_InvalidPhone\_ThrowsException` in our project is a good example, as it prevents corrupt contact records by validating phone numbers.

Finally, avoiding shortcuts helps minimize technical debt. When corners are cut, it often leads to more work later, making it harder to add features and meet deadlines. If we only test valid phone numbers, we might miss issues with invalid data that could complicate future fixes. It's far easier to write comprehensive tests upfront!

Examples:

* Creating Comprehensive Test Coverage: It's super important to write detailed test cases that cover all scenarios, including tricky edge cases. For instance, the test `contactConstructor\_InvalidPhone\_ThrowsException` makes sure that invalid inputs like nulls or non-numeric values get rejected. If we skip these tests, we risk running into bugs later on that could cause problems in production.
* Avoiding Assumptions about Inputs: A disciplined approach to testing means including boundary checks and checking out those weird scenarios, like unusual phone numbers. We can’t just assume that only valid data will be entered, so it’s all about making sure our code is thoroughly validated.