**Summary & Reflections Report: Unit Testing and Testing Techniques**

Alysha Pursley

Southern New Hampshire University

CS320: Software Test, Automation QA

Professor Federico Bermudez

October 18, 2024

**Summary & Reflections Report: Unit Testing and Testing Techniques**

**Summary**

**Unit Testing Approach for Key Features**

For this project, I used JUnit testing to verify the functionality of three key features: Contact, Task, and Appointment. The primary goal was to ensure that each class's fields and methods adhered to the software requirements by validating the input constraints and ensuring the correct functionality of the service methods. Each unit test targeted specific fields such as contactId, firstName, lastName, phoneNumber, and address for the Contact class, taskId, name, and description for the Task class, and *appointmentId*, appointmentDate, and description for the Appointment class. Tests were designed to account for both valid inputs and edge cases, such as invalid field lengths or null values. This approach ensured the classes were robust and reliable across different scenarios.

**Alignment to Software Requirements**

The unit testing approach was meticulously aligned to the software requirements, particularly in validating the constraints and rules defined for each feature. In the Contact feature, the JUnit tests ensured that fields like contactId, firstName, and phoneNumber met specific length and null constraints. For example, the phoneNumber field was required to be exactly 10 digits, and a test was written to verify this, rejecting any number that did not adhere to this requirement:

*/\*\**

*\* Test creating a contact with an invalid phone number length. Ensures that*

*\* an exception is thrown when the phone number is too short.*

*\*/*

@*Test*

*void* *testInvalidPhoneNumberLength*() {

*// Phone number that is too short*

*assertThrows*(*IllegalArgumentException*.*class*, () *->* {

*new* *Contact*(contactId, firstName, lastName, "123", address);

});

}

This alignment ensured that invalid data was not stored in the system, maintaining data integrity.

For the Task feature, the tests were aligned to the requirement that taskId, name, and description fields adhere to specific length constraints. The name field, for instance, was tested to ensure that any input exceeding 20 characters would be rejected, which was a strict project requirement. This test ensured that tasks followed the defined format for valid data:

*/\*\**

*\* Test creating a task with name longer than 20 characters. Expects an*

*\* IllegalArgumentException.*

*\*/*

@*Test*

*void* *testNameCannotExceedMaxLength*() {

*assertThrows*(*IllegalArgumentException*.*class*, () *->* {

*new* *Task*(validTaskId, "This is a very long name that exceeds the limit", validDescription);  *// Name exceeds 20 characters*

});

}

These tests enforced the rules, ensuring that all tasks met the proper length limits and validation constraints.

In the Appointment feature, the software required that appointmentDate be set for future dates and that description must not exceed 50 characters. A unit test was written to ensure that appointments with past dates were rejected, ensuring the validity of scheduled appointments. This alignment directly met the requirement that no past appointments could be created:

*// Test for updating the appointment date with invalid values (past date or null)*

@*Test*

*void* *testSetAppointmentDateInvalid*() {

*Appointment* *appointment* *=* *new* *Appointment*("1234567890", futureDate, "Routine check-up");

*Date* *pastDate* *=* *new* *Date*(*new* *Date*().*getTime*() *-* 86400000); *// 24 hours in the past*

*// Invalid date values should throw exceptions*

*assertThrows*(*IllegalArgumentException*.*class*, () *->* *appointment*.*setAppointmentDate*(pastDate));

*assertThrows*(*IllegalArgumentException*.*class*, () *->* *appointment*.*setAppointmentDate*(null));

}

These tests confirmed that the Appointment class adhered strictly to the software requirements, allowing only valid appointments to be created.

**JUnit Test Coverage Quality**

The quality of the JUnit tests can be defended by the extensive coverage achieved for each feature, ensuring that both expected behavior and edge cases were accounted for. In the Contact feature, the tests covered all the critical operations, such as creating, updating, and deleting contacts. The test for updating the phoneNumber field, for example, was designed to ensure that invalid phone numbers (those not exactly 10 digits) were rejected, demonstrating thorough coverage:

*/\*\**

*\* Test creating a contact with a non-numeric phone number. Ensures that an*

*\* exception is thrown when the phone number contains non-numeric*

*\* characters.*

*\*/*

@*Test*

*void* *testNonNumericPhoneNumber*() {

*// Phone number containing non-numeric characters*

*assertThrows*(*IllegalArgumentException*.*class*, () *->* {

*new* *Contact*(contactId, firstName, lastName, "12345abcde", address);

});

}

This ensured that the Contact class handled both valid and invalid inputs properly, confirming its reliability.

In the Task feature, tests focused on ensuring that every field adhered to its respective constraints and that operations such as task deletion were handled correctly. For instance, the *deleteTask* method was tested to ensure that only the specified task was removed without affecting others. This thorough validation ensured that the TaskService handled tasks correctly:

*/\*\**

*\* Test deleting a task by ID. Ensures that the task is removed from the*

*\* list.*

*\*/*

@*Test*

*void* *testDeleteTask*() {

*taskService*.*addTask*(name, description);

*String* *taskId* *=* *taskService*.*getTaskList*().*get*(0).*getTaskId*();

*taskService*.*deleteTask*(taskId);

*assertTrue*(*taskService*.*getTaskList*().*isEmpty*());

}

These tests provided comprehensive coverage, confirming that the Task class functioned as required.

For the Appointment feature, the tests ensured that the service correctly handled appointment creation, validation, and deletion. The test for adding appointments only allowed those with valid future dates and descriptions that followed the required length constraints. For example, the following test confirms that a description exceeding the character limit results in an *IllegalArgumentException*:

*/\*\**

*\* Test for setting a valid description that exactly meets the maximum length constraint*

*\*/*

@*Test*

*void* *testSetDescriptionMaxLength*() {

*// Create a valid appointment and set a description with exactly 50 characters*

*Appointment* *appointment* *=* *new* *Appointment*("1234567890", futureDate, "Routine check-up");

*String* *maxLengthDescription* *=* "12345678901234567890123456789012345678901234567890"; *// Exactly 50 characters*

*appointment*.*setDescription*(maxLengthDescription); *// Set the max-length description*

*// Assert that the description is correctly set*

*assertEquals*(maxLengthDescription, *appointment*.*getDescription*());

}

This test validates the system's ability to handle invalid input, ensuring the integrity of appointment data.

**Technical Soundness of Code**

To ensure the technical soundness of both the ContactService and TaskService, multiple strategies were applied throughout the development process, focusing on validation and thorough exception handling. This approach was vital in preventing invalid data from entering the system, which helped maintain the robustness and reliability of the code. The implementation of input validation enforced strict constraints on fields such as firstName, lastName, and phoneNumber in the Contact class, as well as name and description in the Task class. This validation guarantees that any invalid input results in an immediate exception, thus upholding the integrity of the data:

*/\*\**

*\* Constructor to initialize a Task object.*

*\**

*\* @param* taskId *the unique ID for this task, must be non-null and up to 10*

*\* characters*

*\* @param* name *the name of the task, must be non-null and up to 20*

*\* characters*

*\* @param* description *the description of the task, must be non-null and up*

*\* to 50 characters*

*\**

*\* This constructor validates all input fields to ensure they meet the*

*\* constraints defined by the class. If any field is invalid, an*

*\* IllegalArgumentException is thrown.*

*\*/*

*public* *Task*(*String* taskId, *String* name, *String* description) {

*// Validate and assign taskId (requirement 1: non-null, not more than 10 characters, cannot be changed)*

*if* (taskId *==* null *||* *taskId*.*length*() *>* MAX\_ID\_LENGTH) {

*throw* *new* *IllegalArgumentException*("Task ID must be non-null and no longer than " *+* MAX\_ID\_LENGTH *+* " characters.");

}

*this*.*taskId* *=* taskId;

This level of input validation is essential for ensuring that the application can handle data correctly.

In addition to input validation, comprehensive exception handling was implemented to manage operations such as updates or deletions where the target entity does not exist. This handling ensures that the operations are both safe and predictable, allowing the system to throw meaningful exceptions for invalid actions. For instance, when attempting to delete a non-existing contact, the system provides a clear indication of the failure:

*/\*\**

*\* Test deleting a non-existent contact.*

*\* Ensures that an exception is thrown when attempting to delete a contact that*

*\* doesn't exist.*

*\*/*

@*Test*

*void* *testDeleteNonExistentContact*() {

*// Test trying to delete a contact that doesn't exist*

*assertThrows*(*IllegalArgumentException*.*class*, () *->* {

*contactService*.*deleteContact*("NonExistentID");

});

}

This proactive approach to error handling contributes significantly to the overall reliability of the application.

Furthermore, edge cases were tested across both the Contact and Task classes to account for scenarios like exceeding maximum allowed lengths or dealing with null values. This rigorous testing confirms that all possible error scenarios are considered, thereby enhancing the technical robustness of the code. By ensuring that each edge case is addressed, the code remains resilient under various conditions:

*/\*\**

*\* Test creating a task with a null name. Expects an*

*\* IllegalArgumentException.*

*\*/*

@*Test*

*void* *testNameCannotBeNull*() {

*assertThrows*(*IllegalArgumentException*.*class*, () *->* {

*new* *Task*(validTaskId, null, validDescription);  *// Name is null*

});

}

Incorporating thorough validation logic and proper exception handling guarantees that the code is technically sound and maintainable.

**Efficiency of the Code**

Efficiency in the ContactService and TaskService classes was achieved by focusing on streamlined logic, minimizing redundancy, and leveraging optimal techniques. By centralizing validation for updating fields like updateFirstName or updatePhoneNumber in the Contact class, as well as updateName in the Task class, the need for repetitive logic was significantly reduced. This design adheres to the single responsibility principle, ensuring that each method is clear and efficient. Such centralization allows for easier maintenance and better consistency across the codebase:

*// Methods to update mutable fields with validation*

*public* *void* *updateFirstName*(*String* firstName) {

*if* (firstName *==* null *||* *firstName*.*length*() *>* MAX\_NAME\_LENGTH) {

*throw* *new* *IllegalArgumentException*("First name must be non-null and no longer than " *+* MAX\_NAME\_LENGTH *+* " characters.");

}

*this*.*firstName* *=* firstName;

}

By implementing such strategies, the code remains organized and efficient.

Additionally, the use of @BeforeEach for efficient test setup ensured that common objects, like taskService and contactService, were initialized only once before all tests. This approach eliminated redundant initialization code in each test method, resulting in a more efficient test execution process. Consequently, tests became easier to maintain and faster to run. Here’s an example illustrating this setup:

@*BeforeEach*

*void* *setUp*() {

*// Initialize the contact service and test data before each test*

contactService *=* *new* *ContactService*();

firstName *=* "John";

lastName *=* "Doe";

phoneNumber *=* "1234567890";

address *=* "123 Main St, Anytown, USA";

}

This setup demonstrates a commitment to efficient testing practices.

Moreover, both classes utilized UUIDs for generating unique identifiers for contacts and tasks, which ensured quick and simple ID generation without unnecessary complexity. By adopting UUIDs, the chances of errors related to duplicate IDs were minimized, enhancing the application's overall performance. Here’s an example of how UUIDs were implemented:

*/\*\**

*\* Generates a new unique ID for a contact. The ID is a 10-character string*

*\* derived from a UUID.*

*\**

*\* @return a unique contact ID string*

*\*/*

*private* *String* *generateUniqueId*() {

*return* *UUID*.*randomUUID*().*toString*().*substring*(0, 10);

    }

This straightforward approach to ID handling significantly contributes to the efficiency of the application while maintaining clarity in the code structure.

**Reflection**

**Software Testing Techniques Employed**

In this project, JUnit testing was the primary technique used to validate the functionality of individual units, such as classes and methods. This approach allowed for isolated testing, meaning each class or method was examined independently to ensure proper behavior. JUnit testing was essential for verifying validation rules in classes like Contact, Task, and Appointment. For example, in the Appointment class, tests confirmed that appointments could only be created if the date were set in the future and that descriptions adhered to character limits. Isolating each unit in this manner ensured the reliability of the code during its fundamental operations.

The tests were designed to check for both valid and invalid inputs, ensuring that any edge cases were handled correctly. TaskService tests confirmed that task descriptions were rejected if they exceeded the specified character limit. These checks were critical for verifying that each service operated according to its specifications. By using JUnit, the tests provided specific validation for individual functions, making it easier to identify and address potential issues before the components were integrated into the broader system.

***Other Software Testing Techniques Not Used***

While JUnit testing was the focus of this project, other important techniques, such as integration testing, were not employed. Integration testing is crucial for ensuring that different system components work together effectively, particularly in larger systems with multiple dependencies. In this project, JUnit’s isolated testing was deemed sufficient since the primary focus was on testing individual units. Regression testing, another method not used, is essential in environments where code is frequently updated to ensure that new changes do not disrupt existing functionality. Similarly, system testing, which evaluates the application in a production-like environment, was not applied in this case.

***Practical Uses and Implications of Testing Techniques***

Each testing technique is suited for specific stages of development and plays a critical role in maintaining software quality. JUnit testing is ideal for the early stages, focusing on the smallest units to ensure a solid foundation. This proactive approach can prevent larger issues from developing by catching problems early in the process. Integration testing becomes vital when individual units must work together, ensuring data flows correctly between different components. Regression testing is particularly crucial in iterative or agile environments, where frequent code changes require vigilance to maintain existing functionality. System testing is essential before deployment, simulating real-world conditions to validate performance, load capacity, and security compliance.

**A Cautious Mindset in Software Testing**

Throughout this project, I adopted a cautious mindset as a software tester, recognizing the significance of examining both common and edge-case scenarios. This caution was particularly evident in the *testContactCreation()* method in the *ContactTest* class, where I validated that the contactId, firstName, and phoneNumber fields adhered to strict requirements. For instance, ensuring that the contactId was non-null and no longer than 10 characters while the phoneNumber consisted of exactly 10 digits was critical for maintaining data integrity. Additionally, understanding the complexity and interrelationships within the system was essential; each class, including Contact, Task, and Appointment, relied on the proper functioning of related services. For example, the AppointmentService class needed rigorous testing of the Appointment class to ensure appointment date validations were correctly implemented. Tests such as testInvalidAppointmentDate() were crucial in preventing past or null dates from being accepted, thereby preserving the integrity of the scheduling system.

***Limiting Bias and Commitment to Quality***

To minimize bias in my code review, I approached the testing process with the mindset of an external reviewer, scrutinizing each method independently. In testing the TaskService class, I addressed potential edge cases, such as ensuring that a null task name would trigger an exception in the *testUpdateTaskNameCannotBeNull()* method. This objective evaluation prevented me from assuming that my code would function as intended. Additionally, I recognized the natural inclination to overlook flaws when testing my own code, especially in the AppointmentService class regarding duplicate appointment IDs. To combat this, I developed the test *testAddDuplicateAppointmentId()* to ensure the service effectively rejected any duplicate IDs. Maintaining a disciplined commitment to quality was vital for preventing technical debt; cutting corners during development can lead to costly fixes later in the software lifecycle. For example, in the Task class, the test *testExceedMaxLengthName()* verified that names longer than 20 characters were appropriately rejected, helping to prevent data overflow issues. By addressing these concerns early on, I avoided accumulating technical debt, which could lead to inefficiencies or failures down the line.

***Practicing Avoidance of Technical Debt***

As a practitioner in the field, I plan to continue adhering to best practices in coding and testing to avoid technical debt. This includes writing comprehensive unit tests that cover various inputs and outputs, ensuring thorough validation before deployment. For example, in the ContactServiceTest, I validated that contact updates were correctly applied with valid inputs. By maintaining diligence during development and addressing potential issues early, I aim to create software that is both reliable and scalable, thereby mitigating the long-term costs associated with technical debt. Ultimately, the combination of a cautious mindset, a focus on limiting bias, and a commitment to quality will guide my efforts in delivering robust software solutions.