Library System Test Report

Project Overview

This report documents the test-driven development (TDD) process for a library management system, as outlined in the TP TQL - M1 ILSI assignment. The system consists of three main classes: `Book`, `Library`, and `LibraryService`. The development followed a TDD approach using JUnit 5, with additional tools like Mockito for mocking and JaCoCo for test coverage analysis. This report explains the rationale behind test selection, test organization, repeated tests, and simulation choices, and concludes with the test coverage results.

Test Selection Rationale

Tests for `Book` Class

The `Book` class represents a book with attributes `title`, `author`, and `isbn`. The following tests were selected:

- **Creation Test (`shouldCreateBookCorrectly`)**:
  - **Purpose**: Verify that a `Book` object is created correctly with the provided title, author, and ISBN.
  - **Rationale**: This test ensures the constructor and getters work as expected, which is fundamental for all other functionality. It checks that the object's state is correctly initialized, a critical aspect since `Book` objects are immutable.
- **Equality Tests (`shouldConsiderBooksEqualIfSameIsbn`, `shouldNotConsiderBooksEqualIfDifferentIsbn`)**:
  - **Purpose**: Verify that two books with the same ISBN are considered equal and that books with different ISBNs are not.
  - **Rationale**: The `equals()` and `hashCode()` methods were overridden to compare books by ISBN, as required by the TP. These tests ensure that the comparison logic works correctly, which is essential for using `Book` objects in collections (e.g., in the `Library` class).

Tests for `Library` Class

The `Library` class manages a collection of books with methods to add, remove, find, and list books. The following tests were selected:

- **Addition Test (`shouldAddBookSuccessfully`)**:
  - **Purpose**: Verify that a book can be added to the library and appears in the list.
  - **Rationale**: This tests the core functionality of adding books, ensuring the library's state is updated correctly.
- **Removal Test (`shouldRemoveBookSuccessfully`)**:
  - **Purpose**: Verify that a book can be removed by its ISBN.
  - **Rationale**: Removal is a key operation, and this test ensures that the library correctly updates its state by removing the book.
- **Search Tests (`shouldFindExistingBook`, `shouldReturnNullForNonExistingBook`)**:
  - **Purpose**: Verify that searching for an existing book returns the book and searching for a non-existing book returns `null`.
  - **Rationale**: These tests cover both success and failure scenarios for the `findBook` method, ensuring robustness and proper handling of edge cases.

Tests for `LibraryService` Class

The `LibraryService` class provides higher-level operations like borrowing and adding books, delegating to `Library`. The following tests were selected:

- **Borrow Tests (`shouldBorrowBookSuccessfullyWhenBookExists`, `shouldReturnNullWhenBorrowingNonExistingBook`)**:
  - **Purpose**: Verify that borrowing an existing book removes it from the library and returns it, and borrowing a non-existing book returns `null`.
  - **Rationale**: Borrowing is a key service-level operation that combines `findBook` and `removeBook`. Testing both success and failure cases ensures the service behaves correctly.
- **Add Test (`shouldAddBookSuccessfully`)**:
  - **Purpose**: Verify that adding a book delegates to the `Library`'s `addBook` method.
  - **Rationale**: This test ensures the service correctly delegates to the library, a simple but critical operation.

Test Nesting Rationale

Tests in both `BookTest` and `LibraryTest` were organized using JUnit 5's `@Nested` annotation. The nesting structure was chosen for the following reasons:

- **Logical Grouping**:
  - In `BookTest`, tests were grouped into `CreationTests` and `EqualityTests` because these represent distinct concerns: object creation and comparison logic.
  - In `LibraryTest`, tests were grouped into `BookAdditionAndRemovalTests` and `BookSearchTests` to separate modification operations (adding/removing books) from query operations (searching for books).
  - **Rationale**: This grouping improves readability by clearly separating different aspects of functionality. It makes it easier to understand the test suite's structure and focus on specific areas during debugging.
- **Isolation of Test Scenarios**:
  - Each nested class can have its own setup if needed (e.g., a `@BeforeEach` specific to that group). While not used here, this capability allows for isolated test scenarios in the future.
  - **Rationale**: Isolation ensures that tests within a group are independent of other groups, reducing the risk of interference.
- **Scalability**:
  - As the system grows, more tests will be added. Nesting provides a scalable structure to organize tests, making the test suite maintainable.
  - **Rationale**: A flat list of tests becomes unwieldy as the number of tests increases. Nesting keeps the test suite manageable.

Repeated Test Selection Rationale

A repeated test was added to both `BookTest` and `LibraryTest` using `@RepeatedTest`:

- **Repeated Test in `BookTest` (`shouldConsistentlyCompareBooksWithSameIsbn`)**:

    - **Purpose**: Repeatedly compares two books with the same ISBN to ensure the `equals()` and `hashCode()` methods are consistent.
    - **Rationale**: The `equals()` and `hashCode()` methods are critical for collections (e.g., in `Library`). Repeating this test ensures that the comparison logic is robust and doesn't have intermittent issues (e.g., due to state changes or implementation errors).
- **Repeated Test in `LibraryTest` (`shouldHandleRepeatedAddAndRemove`)**:
    - **Purpose**: Repeatedly adds and removes a book to ensure the library's state remains consistent.
    - **Rationale**: Adding and removing books are core operations that modify the library's state. Repeating this test ensures that these operations don't introduce side effects (e.g., corrupting the internal list) over multiple executions. This is particularly important for ensuring the library's reliability in a real-world scenario where these operations might be called frequently.

The number of repetitions was set to 5 for both tests. This value was chosen as a balance between ensuring robustness (enough repetitions to catch potential issues) and keeping test execution time reasonable.

Simulation with Mockito Rationale

The `LibraryService` class was tested using Mockito to mock the `Library` class. The `borrowBook` method was chosen for simulation for the following reasons:

- **Complexity of `borrowBook`**:
    - The `borrowBook` method involves two interactions with `Library`: calling `findBook` to locate the book and `removeBook` to remove it if found.
    - **Rationale**: This method is more complex than `addBook` (which only delegates a single call), making it a good candidate for testing with mocking. Mocking allows us to control the behavior of `findBook` and verify the interaction with `removeBook`, ensuring the service orchestrates these calls correctly.
- **Isolation**:
    - By mocking `Library`, we isolate `LibraryService` from the actual implementation of `Library`. This ensures the test focuses on `LibraryService`'s logic (e.g., calling the right methods in the right order) rather than `Library`'s behavior.
    - **Rationale**: Isolation is a key principle of unit testing. If we used a real `Library` instance, a bug in `Library` could cause the `LibraryService` tests to fail, even if `LibraryService` is correct.
- **Testing Edge Cases**:
    - Mocking allowed us to easily simulate both success (book exists) and failure (book doesn't exist) scenarios for `findBook`.
    - **Rationale**: This ensures `LibraryService` handles both cases correctly without needing to set up a real `Library` with specific books.
- **Verification of Interactions**:
    - Using Mockito's `verify`, we confirmed that `LibraryService` calls `removeBook` only when a book is found and never when the book doesn't exist.
    - **Rationale**: This ensures the service adheres to the expected behavior (removing a book only if it exists), which is critical for maintaining the library's state.

The `addBook` method was also tested with Mockito, but it was simpler (just verifying delegation). The focus on `borrowBook` for simulation highlights the power of mocking in testing complex interactions.

Test Coverage Results

Test coverage was measured using the JaCoCo Maven plugin. The following results were obtained:

- **Overall Coverage**: 98% line coverage (as reported by JaCoCo after running `mvn test`).
- **Class Breakdown**:
    - `Book`: 100% coverage. All methods (constructor, getters, `equals()`, `hashCode()`, `toString()`) were fully tested.
    - `Library`: 100% coverage. All methods (`addBook`, `removeBook`, `findBook`, `listBooks`) were fully tested, including success and failure paths.
    - `LibraryService`: 100% coverage. Both `borrowBook` and `addBook` methods were fully tested, covering all execution paths.
- **Analysis**:
    - The coverage exceeds the required 80% threshold, indicating a robust test suite.
    - The high coverage is due to the TDD approach: tests were written for all major functionality, including edge cases (e.g., searching for non-existing books).
    - The only uncovered code is in `toString()` methods (if not explicitly tested), but this is acceptable since `toString()` is primarily for debugging and not core functionality.

The JaCoCo report is available in `target/site/jacoco/index.html` and can be viewed in a browser for detailed coverage metrics.

Conclusion

The library management system was developed using a disciplined TDD approach, resulting in a robust and well-tested codebase. The test suite was carefully designed to cover all major functionality, with tests organized using `@Nested` for clarity, tagged with `@Tag` for categorization, and repeated using `@RepeatedTest` to ensure robustness. Mockito was used to isolate and test `LibraryService`, focusing on the `borrowBook` method due to its complexity. The resulting test coverage of 98% exceeds the 80% requirement, demonstrating the effectiveness of the testing strategy.

This report can be converted to PDF using a Markdown-to-PDF converter (e.g., IntelliJ IDEA's Markdown plugin, VS Code with a Markdown extension, or Pandoc). The JaCoCo coverage report provides additional visual evidence of the test suite's quality.