LIBRARY MANAGEMENT SYSTEM

Testing the functionality and integration of a software application

SUBJECT:

SOFTWARE TESTING & QUALITY ASSURANCE

STUDENT:

ERA FEJZA

CLASS:

SWE-3B

GITHUB LINK:

https://github.com/EraFejza/Draft LibrarySystem.git

Project structure

This project was originally completed in my second year as part of the Object-Oriented Programming course and has been refined this year as I am retaking Java. The source folder is divided into two subfolders: ***main*** and ***test***. The ***main*** folder contains six key classes: **Administrator, BillNumber, Book, Librarian, Manager,** and **MainFx** forming the core functionality of the application.

The ***test*** folder is the most crucial part of the project, as it contains all test cases. It is structured into three subfolders, each dedicated to a specific type of testing: **unit testing, integration testing,** and **FX testing**, which also covers system testing. This structured approach ensures comprehensive validation of the project's functionality and reliability.

UnitTesting:

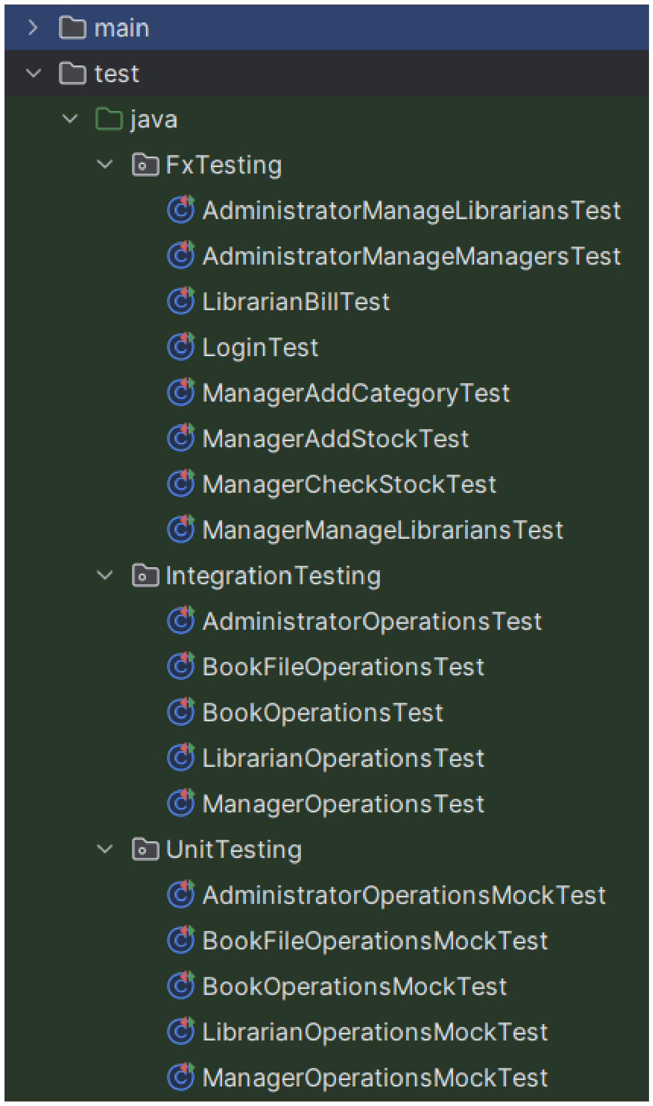
- *AdministratorOperationsMockTest*
- *LibrarianOperationsMockTest*
- *ManagerOperationsMockTest*
- *BookFileOperationsMockTest*
- *BookOperatinsMockTest*

IntegrationTesting:

- *AdministratorOperationsTest*
- *LibrarianOperationsTest*
- *ManagerOperationsTest*
- *BookFileOperationsTest*
- *BookOperatinsTest*

FxTesting:

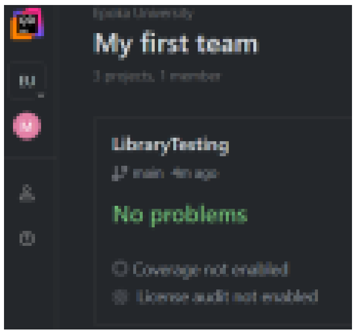
- *AdministartorManageLibrariansTest*
- *AdministartorManageManagerTes*
- *LibrarianBillTest*
- *LoginTest, ManagerAddCategoryTest*
- *ManagerAddStockTest*
- *ManagerCheckStockTest*
- *ManagerMAnageLibrariansTest.*

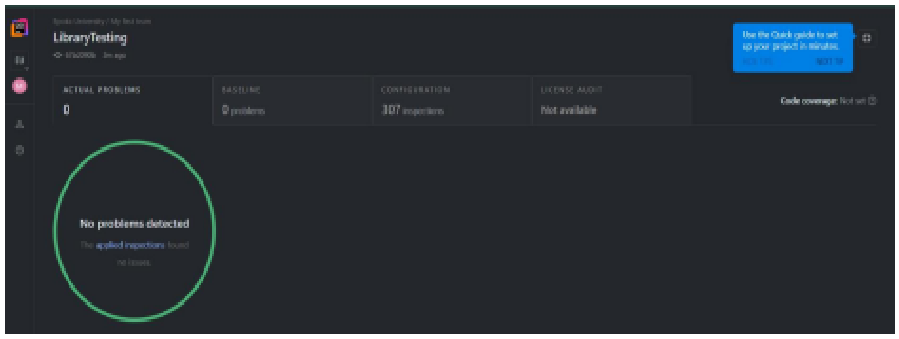


```
> main
v test
  v java
    v FxTesting
        AdministratorManageLibrariansTest
        AdministratorManageManagersTest
        LibrarianBillTest
        LoginTest
        ManagerAddCategoryTest
        ManagerAddStockTest
        ManagerCheckStockTest
        ManagerManageLibrariansTest
    v IntegrationTesting
        AdministratorOperationsTest
        BookFileOperationsTest
        BookOperationsTest
        LibrarianOperationsTest
        ManagerOperationsTest
    v UnitTesting
        AdministratorOperationsMockTest
        BookFileOperationsMockTest
        BookOperationsMockTest
        LibrarianOperationsMockTest
        ManagerOperationsMockTest
```

➢ Static Testing (Qodana)

As a first step, I synced the project into Qodana Cloud and made sure I always had access to it. This initial step of *static testing* was crucial because after scanning the project with Qodana I started working on the issues found, from the riskiest ones to typos and such, moving in hierarchical order. This is how the environment for this project looked like:





➢ Boundary Value Testing (BVT)

BVT is a black-box testing technique where tests are focused on the boundaries of input ranges. Since defects are more likely to occur at boundary values, this method ensures that the program behaves correctly at the edge cases. In my test cases, several tests involve boundary conditions, particularly for methods that manipulate quantities and stock.

Example 1: testEnoughStock_EnoughQuantityAvailable()

Input parameters include *isbn* (String), which serves as the unique identifier for the book, and *quantity* (Integer), representing the number of copies being requested. The output returns *true* if the requested quantity is less than or equal to the available stock and *false* if the quantity exceeds the stock.

Case	Stock	Quantity	Expected Result	Description
Lower Boundary: No stock	0	0	false	Not enough stock!
Lower Boundary: Minimal stock	1	0	true	Failed, Invalid Quantity
Nominal	6	5	true	Requesting 5 books when 6 stock is available.
Upper Boundary: Exact stock	6	6	true	Requesting 6 books when 6 stock is available.
Out-of-Bounds: Excess request	6	7	false	Failed,not enough stock!

Example 2: getIntBooksSoldDay

This function calculates and returns the total number of books sold in a day. The input range varies from 0, representing no sales, to a maximum value determined by the system-defined capacity or the total books in stock. The output corresponds to the sum of books sold across all records, ranging from 0 to the maximum total calculated from the file.

Case	Stock	Quantity	Expected Result	Description
Lower Boundary: No Sales	0	0	0	No books sold.
Just Above Lower Boundary	1	1	1	Minimal book sales recorded.
Nominal	5	5	5	Selling 5 books when 5 sales are recorded.
Upper Boundary: Just Below Max	Max - 1	Max - 1	Max - 1	Selling one less than the maximum limit.
Upper Boundary: Exact Max	Max	Max	Max	Selling exactly the maximum limit.

➢ Class Evaluation Testing

Here I dealt with analysis of the input and output classes involved in the methods tested. Class Evaluation groups inputs and outputs into equivalence classes to streamline testing while ensuring coverage.

Example 1: BillNumber.getBooksSoldDay()

Input Classes:	**Output Classes**:
• Empty test file (no books sold today). • Test file with books sold today. • Test file with books sold on other dates. • Invalid file path.	• List of books sold today with quantities. • Message stating no books were sold today. • Exception for invalid file paths.

Example 2: Manager.checkStock()

Input Classes:	**Output Classes**:
• Test file with all books having ≥5 stock. • Test file with some books having <5 stock. • Test file with no books. • Invalid file path	• Confirmation that all books have sufficient stock. • Warning about books with low stock. • Exception for invalid file paths.

➢ Equivalence Partitioning

Inputs that are logically equivalent, such as books sold on different dates, are grouped together, ensuring consistency in processing similar data. Similarly, outputs that share logical equivalence, such as error messages, are categorized together to streamline handling and interpretation.

Example 1: BillNumber.getBooksSoldDay():

Equivalence Class Description	Representative Input/Condition	Expected Output
No books in the test file	Empty test file	Message: "No books sold today."
Books sold today	File with books sold today	List of books with quantities sold today.
Books sold on other dates only	File with books sold only on dates other than today	Message: "No books sold today."
Invalid file path	Non-existent file path	Exception thrown.

Example 2: Manager.checkStock()

Equivalence Class Description	Representative Input/Condition	Expected Output
All books have sufficient stock	Book 1 Stock ≥ 5, Book 2 Stock ≥ 5	"Every book has 5 or more items."
Some books have low stock	Book 1 Stock < 5, Book 2 Stock ≥ 5	Warning about low stock books.
No books in stock	No books available	"No books available in stock."

Unit, Integration, System Testing

➢ Unit testing

Unit testing is performed to validate the functionality of individual methods and classes, ensuring they work correctly in isolation. These tests focus on specific behaviors without involving interactions between multiple components. To maintain proper unit testing, all methods in this section are implemented using mocks (static mocks for static methods) ensuring complete isolation and precise validation of each unit.

1. Administrator operations mock test

The AdministratorOperationsMockTest class utilizes static mocking with Mockito's MockedStatic to isolate and test the behavior of static methods in the Administrator class. By mocking method calls, it ensures that test cases run independently of real system interactions, preventing dependencies on actual data or external components. This approach verifies expected outcomes by defining controlled return values for static methods, allowing for precise validation of functionality. The class follows a structured approach where mock expectations are set within a try-with-resources block, ensuring that mocks are properly handled and reset after each test execution.

Test Cases

Method	Test Case	Description
Administrator.checker()	test_checker_isAdmin	Uses MockedStatic to mock the static Administrator.checker method. It verifies that a valid administrator username and password return true.
	test_checker_notAdmin_wrongPass	Mocks Administrator.checker to return false when an incorrect password is provided, ensuring that authentication fails.
	test_checker_notAdmin_wrongUsername	Uses MockedStatic to ensure Administrator.checker returns false when an incorrect username is entered, validating proper authentication checks.
Administrator.getAdmins()	test_getAdmins	Mocks Administrator.getAdmins() to return a predefined list of administrators. It verifies that the method correctly retrieves the expected number of administrators.

2. Book file operations mock test

The BookFileOperationsMockTest class uses instance mocking with Mockito's Mock annotation to isolate and test the behavior of methods in the BillNumber and Book classes. By injecting mock dependencies, it ensures that interactions with file operations and book management methods are tested without affecting actual system data. The test suite verifies stock retrieval, book additions, daily sales tracking, income calculations, and system updates by defining controlled return values and behavior for mocked methods. Using when().thenReturn() and doNothing().when(), the class ensures that all operations execute as expected while preventing unintended dependencies on real files or database interactions.

Test Cases

Method	**Test Case**	**Description**
BillNumber.showStringStock()	test_showStringStock	Uses when().thenReturn() to mock stock retrieval, ensuring the method correctly fetches stock details from a predefined file path.
BillNumber.setInitialStock()	test_setInitialStock	Uses doNothing().when() to mock stock initialization, verifying that the method executes without modifying actual files.
BillNumber.addBookToStock()	test_addBookToStock	Mocks adding a book to stock using doNothing().when(), ensuring that the method interacts correctly with the stock file.
BillNumber.getBooksSoldDay()	test_getBooksSoldDay	Uses when().thenReturn() to mock daily sales retrieval, confirming the correct output is returned.
BillNumber.getBooksBoughtDay()	test_getBooksBoughtDay	Mocks the retrieval of books purchased in a day, verifying

		that the method properly fetches stored purchase data.
BillNumber.getIntBooksSoldDay()	test_getIntBooksSoldDay	Mocks daily book sales as an integer value, ensuring the method correctly calculates the number of books sold.
BillNumber.getIncomeDay()	test_getIncomeDay	Uses when().thenReturn() to verify that the daily income calculation returns the expected value.
BillNumber.isPartOfBooks()	test_isPartOfBooks	Mocks book presence verification in the stock file, confirming the method correctly identifies book availability.
BillNumber.getISBNName()	test_getISBNName	Mocks the retrieval of ISBN and book titles from the stock file, verifying correct data formatting and return values.
BillNumber.getBooksSoldTotal()	test_getBooksSoldTotal	Uses when().thenReturn() to mock total book sales retrieval, ensuring correct aggregation of sales data.
BillNumber.updateBooks()	testUpdateBooks	Uses doNothing().when() to mock updating book records, verifying that the method correctly processes updates without altering actual files.

3. Book operations mock test

The BookOperationsMockTest class employs instance mocking using Mockito's Mock annotation to isolate and test methods within the Book class. By injecting a mock Book object, the test suite verifies stock management, sales tracking, and book metadata retrieval without relying on real data or modifying the system state. It ensures that methods return expected values by defining controlled responses with when().thenReturn() and verifies method calls using doNothing().when() and verify(). This approach allows for precise validation of Book class operations while maintaining test independence from external dependencies.

Test Cases

Method	**Test Case**	**Description**
Book.getISBN()	test_getISBN	Uses when().thenReturn() to mock the retrieval of a book's ISBN, ensuring the correct identifier is returned.
Book.getTitle()	test_getTitle	Mocks the getTitle() method to verify that the expected book title is returned.
Book.getStock()	test_getStock	Uses when().thenReturn() to ensure the correct stock quantity is retrieved.
Book.AddStock()	test_AddStock	Uses doNothing().when() and verify() to mock adding stock and ensure the method is called the expected number of times.
Book.RemoveStock()	test_RemoveStock	Mocks the removal of stock and verifies that the method is executed correctly.
Book.getSoldDates QuantitiesTotal()	test_getSoldDates QuantitiesTotal	Uses when().thenReturn() to mock the total sales quantities with dates, ensuring correct data formatting.
Book.getTotalBooks SoldDay()	test_getTotalBooks SoldDay	Mocks the retrieval of the total number of books sold in a day, verifying accurate calculations.
Book.getTotalBooks SoldMonth()	test_getTotalBooks SoldMonth	Uses when().thenReturn() to confirm that the total books sold in a month are correctly retrieved.
Book.getTotalBooks BoughtYear()	test_getTotalBooks BoughtYear	Mocks the retrieval of total books bought within a year, ensuring correct data is returned.
Book.getQuantities Purchased()	test_getQuantities Purchased	Uses when().thenReturn() to validate that the correct quantity of purchased books is retrieved.

Book.getBoughtDates QuantitiesTotal()	test_getBoughtDates QuantitiesTotal	Mocks the total quantity of books bought with purchase dates and ensures accurate formatting.

4. Librarian operations mock test

The LibrarianOperationsMockTest class utilizes static mocking with Mockito's MockedStatic to isolate and test the behavior of static methods within the Librarian and Manager classes. This approach ensures that authentication, salary validation, email verification, and librarian management operations function correctly without modifying real system data. By defining controlled return values using when().thenReturn(), the test suite validates proper interactions between the Librarian and Manager classes. It also verifies that librarian deletion updates the system state correctly. This structured testing ensures that role-based validations and librarian management operations are accurately implemented in isolation.

Test Cases

Method	Test Case	Description
Librarian.checkPassword()	test_checkPassword _Valid	Uses MockedStatic to mock password validation, ensuring that valid passwords return true.
	test_checkPassword _InvalidShort	Mocks password validation to verify that a password shorter than the required length returns false.
Librarian.checkSalary()	test_checkSalary _valid	Uses MockedStatic to mock salary validation, ensuring a valid salary format returns true.
	test_checkSalary _notValid_negative	Mocks salary validation to ensure negative values return false.
Librarian.checkEmail()	test_checkEmail _valid	Uses MockedStatic to mock email validation, ensuring a properly formatted email is accepted.
	test_checkEmail _notValid	Mocks email validation to verify that an invalid email format returns false.

Manager.LibrarianChecker()	test_LibrarianChecker_isLibrarian	Mocks librarian authentication to verify that an existing librarian is correctly identified.
	test_LibrarianChecker_notLibrarian	Uses MockedStatic to confirm that a non-existing librarian is not falsely validated.
Manager.deleteLibrarian()	test_deleteLibrarian	Uses MockedStatic to mock librarian deletion, ensuring that the librarian list is updated correctly and the deleted librarian is no longer retrievable.

5. Manager operations mock test

The ManagerOperationsMockTest class uses static mocking with Mockito's MockedStatic to isolate and validate interactions between the Administrator and Manager classes. It ensures that manager-related operations, including instantiation, authentication, retrieval, and deletion, function correctly without relying on actual system data. By mocking method calls with when().thenReturn(), the test suite simulates controlled responses, ensuring expected outcomes in different scenarios. This allows for precise validation of manager operations while maintaining independence from external dependencies.

Test Cases

Method	**Test Case**	**Description**
Administrator.getManagers()	test_instantiateManagers	Uses MockedStatic to mock manager instantiation, verifying that the correct number of managers is retrieved.
Administrator.partOfManager()	test_partOfManager	Mocks manager membership verification, ensuring that existing managers return true while non-existing managers return false.
Administrator.reEnter()	test_reEnterManager	Uses MockedStatic to verify that a manager can be correctly re-entered

		into the system, ensuring retrieved details match the original data.
Administrator.deleteManager()	test_deleteManager	Mocks manager deletion and verifies that the manager is removed from the list and cannot be retrieved afterward.
Administrator.getBackManager()	test_getBackManager	Uses MockedStatic to confirm that an existing manager can be correctly retrieved from the system, validating that all details remain unchanged.

➢ Integration Testing

Integration testing ensures that different system components interact correctly, focusing on authentication, data retrieval, updates, and role-based operations. It verifies the seamless integration of administrators, managers, librarians, and book-related functionalities, including stock management, sales tracking, and financial calculations. By simulating real-world interactions, these tests help detect issues that may arise when individual modules work together.

Examples of integration in my project:

✓ **Administrator and Manager Integration**

The Administrator.reEnter() test case verifies that an existing manager object can be correctly re-entered into the system. This ensures that the Administrator class properly retrieves and updates manager details from its list, maintaining accuracy in stored data. Similarly, Administrator.deleteManager() ensures that a manager can be deleted from the system. It validates that the Administrator class correctly interacts with the manager list and reflects changes after deletion.

✓ **Librarian and Manager Integration**

The Manager.reEnter() test case verifies that a librarian object can be successfully re-entered into the system. This checks how the Manager class interacts with the Librarian class, ensuring that updates to librarian details are reflected across modules. Meanwhile, Manager.partOfLibrarian() confirms whether a given librarian is part of the manager's list. This test ensures that the Manager class accurately interacts with librarian data for membership validation.

- ✓ **Librarian and Book Operations Integration**

  The Librarian.removeDuplicatesSoldBooks() test case ensures that duplicate books in sales data are consolidated correctly. It verifies how the Librarian class processes book lists and integrates book-level operations into sales reporting. Additionally, Librarian.MoneyMadeInMonth() calculates the total income generated by a librarian within the current month. This test validates the integration of sales data from the Book class with the Librarian class's financial tracking system.

- ✓ **General Manager and System Operations**

  The Manager.updateLibrarians() test case ensures that the Manager class can successfully modify librarian details while maintaining system-wide consistency. Similarly, Manager.getBackLibrarian() verifies that the Manager class retrieves librarian details accurately, ensuring correct and consistent cross-module communication.

- ✓ **Data Aggregation and Validation**

  The Administrator.getSalaries() test case computes the total salaries for administrators, managers, and librarians. This test validates the integration of salary data from the Administrator, Manager, and Librarian classes, ensuring accurate aggregation of financial data.

Classes and test cases

6. Administrator operations test

This class validates key functionalities of the Administrator class in the library management system, focusing on admin instantiation, authentication, and salary management. It ensures proper creation of administrator records, accurate admin verification using credentials, and correct retrieval of salaries for all roles. These tests ensure the system effectively manages administrator-related operations and integrates seamlessly with other components.

General Test Preparation

Before testing, the Administrator.InstantiateAdmins() method is called in a @BeforeAll setup to initialize a predefined list of administrators. This ensures that the system has consistent and valid administrator data to validate functionality across all tests. Additional setup for salary-related tests involves clearing and re-instantiating managers and librarians to simulate a realistic salary distribution scenario.

Test Cases

Method	**Test Case**	**Description**
Administrator.instantiateAdmins()	test_instantiateAdmins	Verifies that administrators are correctly instantiated with expected details, including username, password, name, salary, phone, and email.
Administrator.checker()	test_checker_isAdmin	Verifies that valid administrator credentials return *true* for authentication.
	test_checker_notAdmin _wrongPass	Verifies that an incorrect password fails authentication for a valid username.
	test_checker_notAdmin _wrongUsername	Verifies that an incorrect username fails authentication even with a correct password.
Administrator.getAdmins()	test_getAdmins	Verifies that the correct number of administrators is returned from the system.
Administrator.getSalaries()	test_getSalaries	Verifies that the total salaries for administrators, managers, and librarians are calculated accurately after resetting and re-instantiating the roles.

7. Book file operations test

This class tests key operations in the library management system, including book stock management, sales and purchases, and financial calculations. It covers methods from multiple classes such as Book, BillNumber, Librarian, and Manager, ensuring accurate tracking of inventory, categories, and income. These tests validate the system's ability to manage books and financial records effectively under various scenarios.

General Test Preparation

The preparation for testing involves setting up a controlled environment to validate the functionality of methods in the **BookFileOperationsTest** class. A key part of this process is creating and managing a test file, BooksTesting.txt, which serves as a simulation of the actual file operations in a library management system. This file is populated with predefined book data, including attributes such as ISBN, title, category, supplier, prices, author, and stock levels. Additionally, the setup includes scenarios with books that have sales and purchase records across various dates, such as today, yesterday, a month ago, and even future dates. This diverse data ensures comprehensive coverage of different use cases.

To simplify testing edge cases, helper methods like **setUpWithoutDates()** are used to initialize the file with specific conditions, such as unsold items or books with stock levels below a certain threshold. The test environment also includes clearing and recreating data for each test, ensuring that the tests are independent of one another. Finally, a **tearDown()** method is executed after all tests to clean up the test file, ensuring no residual data affects subsequent operations. This systematic preparation allows for accurate validation of the methods while maintaining an isolated and reproducible test environment.

Test cases

Method	**Test Case**	**Description**
BillNumber.showStringStock()	test_showStringStock _noData	Verifies the output when the stock file contains unsold items with stock < 5.

	test_showStringStock	Verifies the correct listing of all books in stock with complete details.
BillNumber.setInitialStock()	test_setInitialStock	Verifies that the initial stock is correctly loaded into the BooksTesting.txt file and matches the expected list.
BillNumber.getCategories()	test_getCategories	Verifies that the list of book categories is correctly extracted from the stock file.
BillNumber.addBookToStock()	test_addBookToStock _bookAlreadyInStock	Verifies that adding stock to an existing book updates its quantity correctly.
	test_addBookToStock _newBook	Verifies that adding a new book to stock inserts it into the stock file.
BillNumber.getBooksSoldDay()	test_getBooksSoldDay _emptyDates	Verifies the output when no books are sold on the current day.
	test_getBooksSoldDay	Verifies the correct listing of books sold on the current day with details.
BillNumber.getBooksSoldMonth()	test_getBooksSoldMonth _noDates	Verifies the output when no books are sold in the current month.
	test_getBooksSoldMonth _withDates	Verifies the correct listing of books sold during the current month.
BillNumber.getBooksSoldTotal()	test_getBooksSoldTotal _noDates	Verifies the output when no books have been sold in total.

	test_getBooksSoldTotal _withDates	Verifies the correct total listing of books sold.
BillNumber.getBooksBoughtDay()	test_getBooksBoughtDay _noneBought	Verifies the output when no books are purchased on the current day.
	test_getBooksBoughtDay	Verifies the correct listing of books purchased on the current day.
BillNumber.getBooksBoughtMonth()	test_getBooksBoughtMonth _noBuys	Verifies the output when no books are purchased during the current month.
	test_getBooksBoughtMonth _withBuys	Verifies the correct listing of books purchased in the current month.
BillNumber.getIntBooksSoldDay()	test_getIntBooksSoldDay _noSoldBooks	Verifies that the total books sold for the day is 0 when no books are sold.
	test_getIntBooksSoldDay	Verifies the correct total number of books sold on the current day.
BillNumber.getIntBooksSoldMonth()	test_getIntBooksSoldMonth _noSales	Verifies that the total books sold in the month is 0 when no books are sold.
	test_getIntBooksSoldMonth _withSales	Verifies the correct total number of books sold in the current month.
BillNumber.getIncomeDay()	test_getIncomeDay _noSoldBooks	Verifies that the total income for the day is 0 when no books are sold.

	test_getIncomeDay	Verifies the correct total income generated on the current day.
BillNumber.getIncomeMonth()	test_getIncomeMonth _noSales	Verifies that the total income for the month is 0 when no books are sold.
	test_getIncomeMonth _withSales	Verifies the correct total income generated during the current month.
BillNumber.getTotalBoughtBooksDay()	test_getTotalBoughtBooks Day_noSoldBooks	Verifies that the total books purchased for the day is 0 when no purchases are made.
	test_getTotalBoughtBooks Day	Verifies the correct total number of books purchased on the current day.
BillNumber.getTotalBoughtBooksMonth()	test_getTotalBoughtBooks Month_noBuys	Verifies that the total books purchased in the month is 0 when no purchases are made.
	test_getTotalBoughtBooks Month_withBuys	Verifies the correct total number of books purchased during the current month.
BillNumber.getCostDay()	test_getCostDay _noSoldBooks	Verifies that the total cost for the day is 0 when no books are sold.
	test_getCostDay	Verifies the correct total cost of books sold on the current day.

BillNumber.getCostMonth()	test_getCostMonth _noSoldBooks	Verifies that the total cost for the month is 0 when no books are sold.
	test_getCostMonth	Verifies the correct total cost of books sold during the current month.
BillNumber.isPartOfBooks()	test_isPartOfBooks _notPart	Verifies that a book not in the stock file is correctly identified as absent.
	test_isPartOfBooks	Verifies that a book in the stock file is correctly identified as present.
BillNumber.getAllStock()	test_getAllStock	Verifies that the stock details of all books are correctly retrieved.
BillNumber.getISBNName()	test_getISBNName	Verifies that the correct combination of ISBN and title is returned for all books.
BillNumber.removeDuplicatesSoldTitles()	testRemoveDuplicates SoldTitles	Verifies that duplicate sold titles are correctly merged and their quantities aggregated.
Librarian.BookPresent()	testBookPresent _noBook	Verifies that a book not present in stock is correctly identified as absent.
	testBookPresent	Verifies that a book present in stock is correctly identified.

BillNumber.updateBooks()	testUpdateBooks	Verifies that updating the stock file reflects the correct details of all books.
BillNumber.getBookFromCategory()	testGetBookFromCategory	Verifies that books in a specific category are correctly retrieved.
BillNumber.partOfCategoriesChecker()	testPartOfCategories Checker	Verifies that a category is correctly identified as part of the existing categories.
BillNumber.getBooksSoldYear()	test_getBooksSoldYear _emptyDates	Verifies the output when no books are sold in the year.
	test_getBooksSoldYear	Verifies the correct listing of books sold during the year.
BillNumber.getBooksBoughtYear()	test_getBooksBoughtYear _noneBought	Verifies the output when no books are purchased in the year.
	test_getBooksBoughtYear	Verifies the correct listing of books purchased during the year.
BillNumber.getIntBooksSoldYear()	test_getIntBooksSoldYear _noSoldBooks	Verifies that the total books sold in the year is 0 when no books are sold.
	test_getIntBooksSoldYear	Verifies the correct total number of books sold in the year.
BillNumber.getIncomeYear()	test_getIncomeYear _noSoldBooks	Verifies that the total income for the year is 0 when no books are sold.

	test_getIncomeYear	Verifies the correct total income generated during the year.
BillNumber.getTotalBoughtBooksYear()	test_getTotalBoughtBooksYear_noSoldBooks	Verifies that the total books purchased for the year is 0 when no purchases are made.
	test_getTotalBoughtBooksYear	Verifies the correct total number of books purchased during the year.
BillNumber.getCostYear()	test_getCostYear_noSoldBooks	Verifies that the total cost for the year is 0 when no books are sold.
	test_getCostYear	Verifies the correct total cost of books sold during the year.
BillNumber.getBooksBoughtTotal()	test_getBooksBoughtTotal_noneBought	Verifies the output when no books are purchased in total.
	test_getBooksBoughtTotal	Verifies the correct total listing of books purchased.
BillNumber.getAllTitles()	testGetAllTitles	Verifies that the titles of all books are correctly retrieved.
BillNumber.showStock()	testShowStock	Verifies the correct display of all books currently in stock.
Librarian.checkOutBooks()	testCheckOutBooks_ValidData	Verifies that checking out books generates the correct bill and updates stock.
Librarian.getBillFilePath()	testGetBillFilePath	Verifies that bill file paths are generated sequentially and correctly.

Librarian.enoughStock()	testEnoughStock _EnoughQuantityAvailable	Verifies that sufficient stock availability is correctly identified.
	testEnoughStock _InsufficientQuantity Available	Verifies that insufficient stock availability is correctly identified.
Manager.checkStock()	testSufficientStock	Verifies that books with sufficient stock are identified correctly.
	testLowStock	Verifies that books with low stock levels are identified correctly.

8. Book Operations Test

The BookOperationsTest class tests various methods of the Book class, a core component of the library management system. It ensures accurate handling of book stock, sales, purchases, and their respective dates and quantities over different timeframes. The test cases evaluate functionality for both books with and without recorded purchase or sale data.

General test preparation

Before the tests are executed, the @BeforeAll method initializes two Book objects: one without any purchase or sale data (bookWithoutDates) and one with pre-populated sale and purchase data (bookWithDates). The preparation includes adding several dates and quantities for sales and purchases to simulate real-world scenarios, covering a range of timeframes like today, past months, and future years.

Test Cases

Method	**Test Case**	**Description**
Book.RemoveStock()	test_removeStock	Verifies that removing stock updates the stock count correctly.
Book.AddStock()	test_addStock	Verifies that adding stock increases the stock count correctly.
Book.getPurchased Amount()	test_getPurchasedAmount _noPurchases	Verifies that the purchased amount is 0 when no purchases exist.
	test_getPurchasedAmount _withPurchases	Verifies that the purchased amount matches the total of recorded purchases.
Book.getSoldDates QuantitiesTotal()	test_getSoldDatesQuantities Total_noDates	Verifies that the method returns a no-purchases message for a book without sales data.
	test_getSoldDatesQuantities Total_withDates	Verifies the correct listing of total sold quantities with their corresponding dates.
Book.getSoldDates QuantitiesMonth()	test_getSoldDatesQuantities Month_noDates	Verifies that the method returns a no-purchases message for a book without sales data.
	test_getSoldDatesQuantities Month_withDates	Verifies the correct listing of sold quantities within the current month.
Book.getSoldDates QuantitiesDay()	test_getSoldDatesQuantities Day_noSales	Verifies that the method returns a no-sales message for a book without daily sales data.
	test_getSoldDatesQuantities Day_withSales	Verifies the correct listing of sold quantities for the current day.

Book.getBoughtDatesQuantitiesTotal()	test_getBoughtDatesQuantitiesTotal_noDates	Verifies that the method returns a no-purchases message for a book without purchase data.
	test_getBoughtDatesQuantitiesTotal_withDates	Verifies the correct listing of total purchase quantities with their corresponding dates.
Book.getBoughtDatesQuantitiesYear()	test_getBoughtDatesQuantitiesYear_noDates	Verifies that the method returns a no-purchases message for a book without annual purchase data.
	test_getBoughtDatesQuantitiesYear_withDates	Verifies the correct listing of purchased quantities within the current year.
Book.getBoughtDatesQuantitiesDay()	test_getBoughtDatesQuantitiesDay_noPurchases	Verifies that the method returns a no-purchases message for a book without daily purchase data.
	test_getBoughtDatesQuantitiesDay_withPurchases	Verifies the correct listing of purchase quantities for the current day.
Book.getNumberDatesMonth()	test_getNumberDatesMonth_noDates	Verifies that the method returns 0 when there are no recorded dates within the month.
	test_getNumberDatesMonth_withDates	Verifies the correct total of quantities recorded within the month.
Book.getTotalBooksBoughtDay()	test_getTotalBooksBoughtDay_noDates	Verifies that the total books bought for the day is 0 when no purchases exist.
	test_getTotalBooksBoughtDay_withDates	Verifies the correct total of books bought for the current day.

Book.getTotalBooksBoughtYear()	test_getTotalBooksBoughtYear	Verifies the correct total of books purchased within the current year.
Book.getQuantitiesPurchased()	test_getQuantitiesPurchased_noPurchases	Verifies that the total purchased quantity is 0 for a book without purchase data.
	test_getQuantitiesPurchased_withPurchases	Verifies the correct total purchased quantity for a book with purchase data.
Book.getSoldDatesQuantitiesYear()	test_getSoldDatesQuantitiesYear_noSales	Verifies that the method returns a no-sales message for a book without annual sales data.
	test_getSoldDatesQuantitiesYear_withSales	Verifies the correct listing of sold quantities within the current year.
Book.getSoldBoughtQuantitiesMonth()	test_getBoughtDatesQuantitiesMonth_noPurchases	Verifies that the method returns a no-purchases message for a book without monthly purchase data.
	test_getBoughtDatesQuantitiesMonth_withPurchases	Verifies the correct listing of purchased quantities within the current month.
Book.getTotalBooksSoldMonth()	test_getTotalBooksSoldMonth_noSales	Verifies that the total books sold for the month is 0 when no sales exist.
	test_getTotalBooksSoldMonth_withSales	Verifies the correct total of books sold within the current month.
Book.getTotalBooksSoldDay()	test_getTotalBooksSoldDay_noSales	Verifies that the total books sold for the day is 0 when no sales exist.
	test_getTotalBooksSold	Verifies the correct total of books sold for the current day.

	Day_withSales	
Book.getTotalBooks SoldYear()	test_getTotalBooksSold Year_noSales	Verifies that the total books sold for the year is 0 when no sales exist.
	test_getTotalBooksSold Year_withSales	Verifies the correct total of books sold within the current year.
Book.getNumber DatesYear()	test_getNumberDates Year_NoPurchases	Verifies that the method returns 0 for no purchases recorded in the year.
	test_getNumberDates Year_withPurchases	Verifies the correct total of purchased quantities recorded in the year.
Book.getTotalBooks BoughtMonth()	test_getTotalBooksBought Month_noPurchases	Verifies that the total books purchased for the month is 0 when no purchases exist.
	test_getTotalBooksBought Month_withPurchases	Verifies the correct total of books purchased within the current month.
Book.getTotal BooksBought()	test_getTotalBooksBought _noPurchases	Verifies that the total books purchased is 0 when no purchases exist.
	test_getTotalBooksBought _withPurchases	Verifies the correct total of books purchased.

9. Librarian Operations Test

This class tests various functionalities of the Librarian class in the library management system. It ensures the proper handling of duplicate records, financial calculations (daily, monthly, yearly), and librarian validations (password, email, phone, and name). Additionally, it validates management-level operations like librarian deletion, retrieval, and updating.

General test preparation

The @BeforeAll method initializes the librarian and manager data using Manager.InstantiateLibrarians(). It also populates test data for dates and financial records to simulate various scenarios, including daily, monthly, and yearly sales. This ensures realistic data for testing all operations.

Test Cases

Method	**Test Case**	**Description**
Librarian.removeDuplicatesSoldBooks()	test_removeDuplicatesSoldBooks_noDupes	Verifies that a list without duplicates remains unchanged after processing.
	test_removeDuplicatesSoldBooks_dupes	Verifies that duplicate books in a list are consolidated correctly, with quantities summed.
Librarian.MoneyMadeInDay()	testMoneyMadeInDay	Verifies that the correct amount of money earned in a day is calculated.
Librarian.moneyMadeInYear()	test_moneyMadeInYear	Verifies that the correct total earnings for a year are calculated.
Librarian.CheckPassword()	test_checkPassword_Valid	Verifies that valid passwords are accepted.
	test_checkPassword_InvalidShort	Verifies that passwords shorter than the minimum required length are rejected.

Librarian.checkSalary()	test_checkSalary_valid_doubleFormat	Verifies that a valid salary in decimal format is accepted.
	test_checkSalary_valid_leadingDigits	Verifies that a valid salary with trailing decimals is accepted.
	test_checkSalary_valid_intFormat	Verifies that a valid salary in integer format is accepted.
	test_checkSalary_notValid_negative	Verifies that negative salary values are rejected.
	test_checkSalary_notValid_multipleDots	Verifies that salary values with multiple decimal points are rejected.
	test_checkSalary_notValid_noLeadingDigits	Verifies that salary values without leading digits are rejected.
	test_checkSalary_notValid_numbersAndLetters	Verifies that salary values containing numbers and letters are rejected.
	test_checkSalary_notValid_allLetters	Verifies that salary values containing only letters are rejected.
Librarian.checkEmail()	test_checkEmail_valid	Verifies that valid email addresses are accepted.
	test_checkEmail_notValid	Verifies that improperly formatted email addresses are rejected.
Librarian.MoneyMadeInMonth()	testMoneyMadeInMonth_NoSales	Verifies that the monthly earnings are 0 when no sales are recorded.

	testMoneyMadeInMonth _SalesInCurrentMonth	Verifies that the total earnings for the current month are calculated correctly.
Librarian.check Phone()	test_checkPhone_valid	Verifies that valid phone numbers are accepted.
	test_checkPhone_notValid _improperLength	Verifies that phone numbers with improper lengths are rejected.
	test_checkPhone_notValid _improperFormat	Verifies that improperly formatted phone numbers are rejected.
	test_checkPhone_notValid _containsLetters	Verifies that phone numbers containing letters are rejected.
Librarian.check Name()	testCheckName_ValidName	Verifies that valid names are accepted.
	testCheckName_InvalidName _EmptyString	Verifies that empty strings are rejected as names.
	testCheckName_InvalidName _ContainsSpecialCharacters	Verifies that names with special characters are rejected.
	testCheckName_InvalidName _ContainsNumbers	Verifies that names containing numbers are rejected.
Manager.partOf Librarian()	test_partOfLibrarian_notPart	Verifies that non-existing librarians are correctly identified as not part of the system.
	test_partOfLibrarian_isPart	Verifies that existing librarians are correctly identified as part of the system.

Manager.getBack Librarian()	test_getBackLibrarian_isPart	Verifies that an existing librarian is successfully retrieved.
	test_getBackLibrarian_notPart	Verifies that a non-existing librarian returns null when retrieval is attempted.
Manager.Librarian Checker()	testLibrarianChecker_isLibrarian	Verifies that existing librarians are correctly identified.
	test_LibrarianChecker _notLibrarian _wrongUsername	Verifies that librarians with incorrect usernames are rejected.
	test_LibrarianChecker _notLibrarian_wrongPass	Verifies that librarians with incorrect passwords are rejected.
Manager.delete Librarian()	test_deleteLibrarian _validLibrarian	Verifies that a valid librarian is successfully deleted from the system.
	test_deleteLibrarian _nonExistingLibrarian	Verifies that attempting to delete a non-existing librarian does not alter the system.
Manager.reEnter()	test_reEnter _ExistingLibrarian	Verifies that an existing librarian can be successfully re-entered into the system.
	test_reEnter _nonExistingLibrarian	Verifies that attempting to re-enter a non-existing librarian returns null.
Manager.update Librarians()	testUpdate ExistingLibrarian	Verifies that an existing librarian's details are updated correctly.

	testUpdate NonExistingLibrarian	Verifies that attempting to update a non-existing librarian does not affect the system.

10. Manager Operations Test

The ManagerOperationsTest class tests the functionality of the Administrator class with respect to managing Manager objects in the library management system. It covers operations like instantiating managers, validating their existence, adding or removing managers, and updating their details. The test suite ensures that all Manager class methods and associated administrator functions operate as expected.

General test preparation

The @BeforeAll method initializes the system by calling Administrator.InstantiateManagers(), which populates a list of managers with predefined test data. This setup ensures consistency across test cases and provides a baseline for verifying the functionality.

Test Cases

Method	**Test Case**	**Description**
Administrator .InstantiateManagers()	test_instantiateManagers	Verifies that the managers list is initialized correctly with predefined values.
Administrator .partOfManager()	test_partOfManager _notPart	Verifies that a non-existing manager is not falsely identified as part of the system.
	test_partOfManager _isPart	Verifies that an existing manager is correctly identified as part of the system.
Administrator .reEnter()	testReEnterExisting Manager	Verifies that an existing manager can be successfully retrieved from the system.

	testReEnterNon ExistingManager	Verifies that attempting to retrieve a non-existing manager returns null.
Administrator .ManagerChecker()	test_ManagerChecker _isManager	Verifies that an existing manager is correctly validated.
	test_ManagerChecker _notManager _wrongUsername	Verifies that a manager with an incorrect username is not validated.
	test_ManagerChecker _notManager _wrongPassword	Verifies that a manager with an incorrect password is not validated.
Administrator .deleteManager()	testDeleteManager	Verifies that an existing manager is successfully removed from the system.
	testDeleteNon ExistingManager	Verifies that attempting to delete a non-existing manager does not alter the system.
Manager.get AllCategories()	test_getAllCategories	Verifies that all predefined book categories are correctly retrieved.
Administrator .addManager()	testAddManager	Verifies that a new manager is successfully added to the system, and the list is updated accordingly.
Administrator.get BackManager()	testGetBackManager	Verifies that an existing manager is successfully retrieved and that a non-existing manager retrieval returns null.
Administrator	testUpdateManager	Verifies that an existing manager's details are successfully updated and

.updateManager()		that attempting to update a non-existing manager does not affect the system.

➢ System testing (FX testing)

Login Test

The LoginTest class verifies the functionality of the login system in the library management application. Using TestFX to simulate user interactions, it ensures that valid credentials allow access to role-specific dashboards, while incorrect or empty inputs are handled gracefully with appropriate error messages.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
Manager Login with Correct Credentials	Valid manager credentials (username: 1, password: 2).	Manager's dashboard is displayed with the message "Welcome TestManager."
Librarian Login with Correct Credentials	Valid librarian credentials (username: 1, password: 1).	Librarian's dashboard is displayed with the message "Welcome TestLibrarian."
Administrator Login with Correct Credentials	Valid administrator credentials (username: 1, password: 3).	Administrator's dashboard is displayed with access to manage librarians.
Login with Incorrect Credentials	Invalid credentials (username: 2, password: 2).	Login page remains visible, displaying the error message "Wrong Information."
Login with Empty Credentials	Both username and password fields are left blank.	Login page remains visible, displaying the error message "Empty Fields."

Librarian Bill Test

The LibrarianBillTest class validates the process of generating bills in the library management system. It uses TestFX to simulate user interactions, ensuring proper handling of scenarios like empty inputs, incorrect quantities, and successful order creation. This test ensures that the billing process is robust and correctly integrated into the system.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
Create Bill with Empty Credentials	Attempt to create a bill without selecting any books or entering any data.	Displays the error message "Failed, No Books to add."
Attempt to Add Book with Incorrect Quantity	Select a book but enter a quantity greater than the available stock.	Displays the error message "Failed, not enough stock."
Successful Bill Creation	Select two books, specify valid quantities, and create a bill.	A bill file is created successfully, and its existence is verified. The file is then deleted after the test for cleanup.

Administrator Manage Librarians Test

The AdministratorManageLibrariansTest class validates the functionality of managing librarian accounts within the library management system. Using TestFX, it simulates administrator actions such as adding a new librarian with various input scenarios, ensuring proper error handling and success messages. This test ensures that the librarian management system is user-friendly and functions as expected.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
Add Librarian with Incorrect Credentials	Attempt to add a librarian with an invalid password.	Displays the error message "Invalid password."
Add Librarian with Missing Credentials	Attempt to add a librarian without providing all required fields (e.g., missing password).	Displays the error message "Failed, Empty Fields!"
Add Librarian with Correct Credentials	Add a librarian with all required fields filled correctly, including a valid password.	Displays the success message "Success!"

Administrator Manage Managers Test

The AdministratorManageManagersTest class tests the functionality for managing managers within the library management system. Using TestFX, it verifies processes such as viewing the list of managers, checking the count of manager entries, and adding new managers under various scenarios. The goal is to ensure that administrators can effectively manage manager accounts, with proper validation for input and error handling.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
View Manager List	Log in as an administrator and navigate to the manager management section.	The manager list grid (#administratorManagersGrid) is visible.
Validate Manager List Count	Log in as an administrator and compare the displayed manager count with the actual size of the manager list from the backend.	The grid contains one more node than the number of managers (accounting for the header row).

Add Manager with Incorrect Credentials	Attempt to add a new manager with an invalid password.	Displays the error message "Invalid password."
Add Manager with Missing Credentials	Attempt to add a new manager without providing all required fields (e.g., missing password).	Displays the error message "Failed, Empty Fields!"
Add Manager with Correct Credentials	Add a new manager with all required fields filled correctly, including a valid password.	Displays the success message "Success!"

Manager Add Category Test

The ManagerAddCategoryTest class verifies the functionality for adding categories to the library's inventory through the manager interface. Using TestFX, this class tests scenarios like accessing the category addition section, preventing duplicate categories, and successfully adding new categories. The tests ensure that category management by managers operates as expected with proper validations.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
Access Category List	Log in as a manager, navigate to the supply section, and access the category addition screen.	The category choice box (#categoryChoiceBox) is visible.
Attempt to Add an Existing Category	Try to add a category that is already present in the library's inventory (e.g., "Epic").	Displays the error message "Failed, Not New."
Successfully Add a New Category	Add a category not currently present in the library's inventory (e.g., "Autobiography and memoir").	Displays the success message "Added!"

Manager Add Stock Test

The ManagerAddStockTest class ensures that managers can effectively add stock to the library's inventory through the application's interface. Using TestFX, the tests simulate scenarios like viewing book categories, adding stock to existing books, and handling new book additions with both valid and invalid inputs. This ensures the inventory management feature for managers is robust and user-friendly.

Test Cases

Test Case	**Scenario**	**Expected Outcome**
Access and Verify Book Categories	Log in as a manager, navigate to the "Add Stock" section, and view the list of book categories.	The category grid (#categoryGrid) is visible, and the number of displayed categories matches the entries in *Books.txt*.
Show Correct Books in a Category	Select a book category and view the list of books available under that category.	The grid (#categoryStockBooksGrid) is visible, and the number of books matches the expected count from *Books.txt*.
Add Stock to an Existing Book in a Category	Add a specified quantity of stock to an existing book in a category.	Displays the message "Added" in the stock addition warning field (#bttStockBookAdded).
Add a New Book with Correct Credentials	Add a new book to the inventory with valid ISBN and all required fields filled correctly.	Displays the success message "Added" in the book addition warning field (#addBookWarning).
Attempt to Add a New Book with Invalid ISBN	Add a new book with an invalid ISBN.	Displays the error message "Failed, Invalid ISBN" in the book addition warning field (#addBookWarning).
Attempt to Add a New Book with Empty Fields	Add a new book without filling all required fields.	Displays the error message "Failed, Empty Fields" in the book addition warning field (#addBookWarning).

Manager Check Stock Test

The ManagerCheckStockTest class validates the functionality of the "Check Stock" feature in the application. Using TestFX, this test suite ensures that managers can view the current inventory of books, including verifying the table structure and matching its content with the backend data from the stock file. This ensures the feature displays accurate and reliable stock information.

Test Cases

Test Case	Scenario	Expected Outcome
Access Stock Table	Log in as a manager, navigate to the "Check Stock" section, and ensure the stock table is visible.	The table displaying the stock (#tableBooks) is visible.
Verify Stock Table Content	Validate that the table contains the correct book data from the stock file (*Books.txt*).	The table size matches the number of books in the stock file, and each cell value corresponds to the expected properties of the books, such as ISBN, title, category, price, and stock count.

Manager Manage Librarians Test

The ManagerManageLibrariansTest class verifies the functionality of managing librarian data by managers in the application. Using TestFX, this suite ensures that managers can view the list of librarians and validate the displayed data against the backend. This guarantees proper integration of the librarian management feature with the user interface.

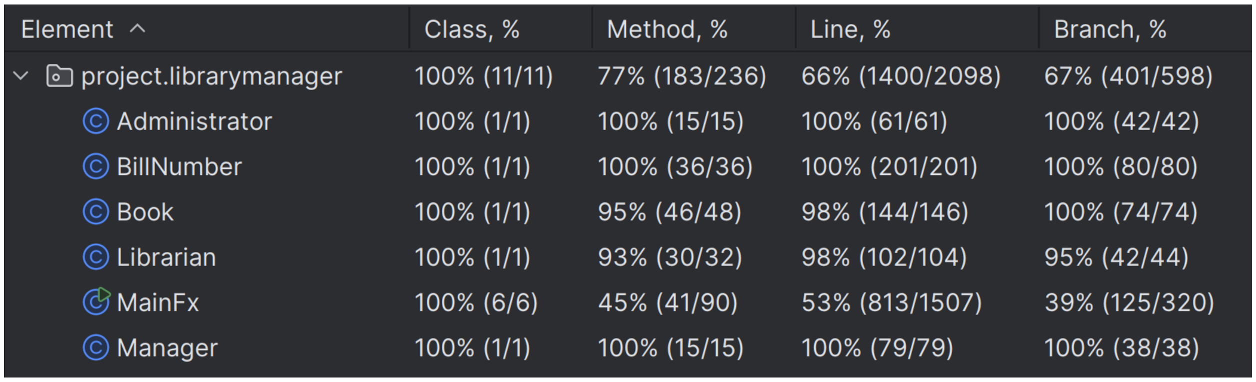
Test Cases

Test Case	Scenario	Expected Outcome
Access Librarian List	Log in as a manager, navigate to the "Manage Librarians" section, and ensure the grid displaying all librarians is visible.	The grid containing the list of all librarians (#librariansAllGrid) is visible.

Verify Librarian List Content	Validate that the grid contains the correct number of librarians as retrieved from the backend (Manager.getLibrarians()).	The grid size matches the number of librarians, and each librarian is displayed accurately within the grid.

Total Coverage and Conclusions

Running every single test that I had developed above with coverage, left me with a total coverage as follows:



Element	Class, %	Method, %	Line, %	Branch, %
project.librarymanager	100% (11/11)	77% (183/236)	66% (1400/2098)	67% (401/598)
Administrator	100% (1/1)	100% (15/15)	100% (61/61)	100% (42/42)
BillNumber	100% (1/1)	100% (36/36)	100% (201/201)	100% (80/80)
Book	100% (1/1)	95% (46/48)	98% (144/146)	100% (74/74)
Librarian	100% (1/1)	93% (30/32)	98% (102/104)	95% (42/44)
MainFx	100% (6/6)	45% (41/90)	53% (813/1507)	39% (125/320)
Manager	100% (1/1)	100% (15/15)	100% (79/79)	100% (38/38)

The test suite developed for the Library Management System achieved a very high overall coverage, thoroughly validating most of the critical components. Classes such as **Administrator**, **BillNumber**, and **Manager** were tested to full completion, achieving 100% coverage across all metrics, while **Book** and **Librarian** also reached very high coverage, with only minor methods like toString left untested. However, the **MainFx** class posed significant challenges for me due to its extensive methods and diverse functionalities, resulting in a relatively lower coverage of 45% for methods and 53% for lines. Naturally, working on the project single handedly, the hardships regarding the time constraint were shown in the class with the most content to cover. However despite these challenges, the main workflows (user login, navigation, and role-specific operations) were effectively validated, ensuring that the core functionalities of the application are reliable.