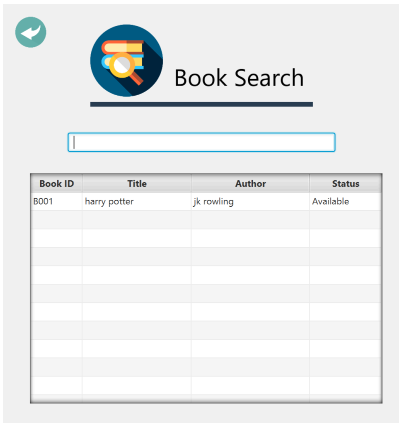
Library Management System SWE303:

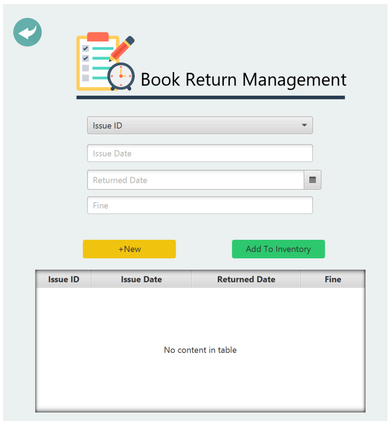
Layan Abdulaziz Alateeq & Klea Faqoli

About Our Project

Our project describes a library management system with Javafx features, it allows the addition, updating, deletion, and searching of books, our system also manages members, adding, deleting, and updating their details, the program also manages the issuing and returns of books regarding certain members. It uses SQL as its form of a database and uses an MVC structure.

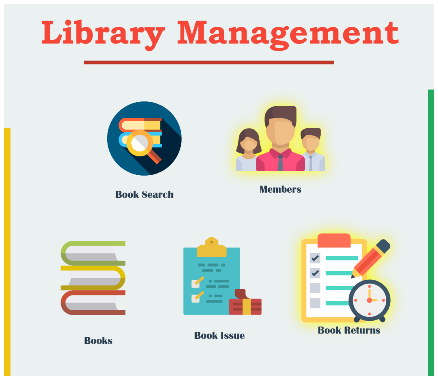


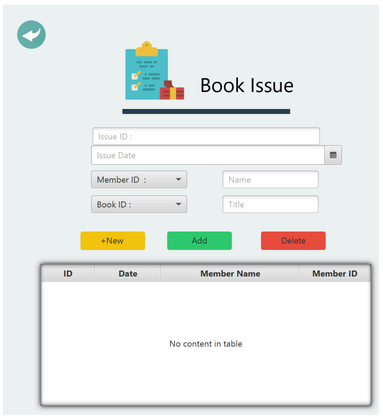
Book ID	Title	Author	Status
B001	harry potter	jk rowling	Available



Issue ID	Issue Date	Returned Date	Fine

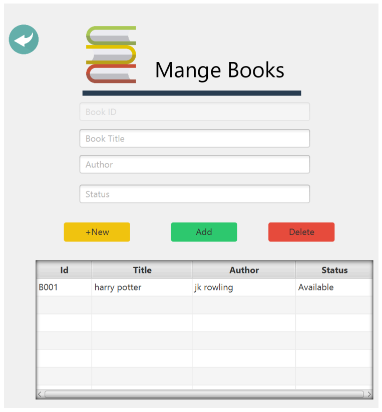
No content in table



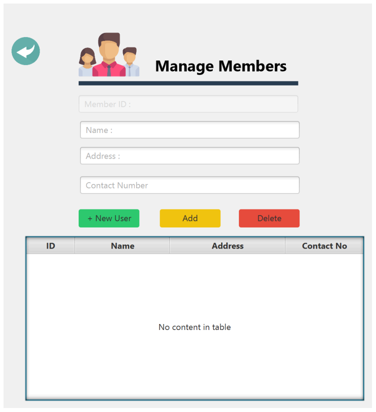


ID	Date	Member Name	Member ID

No content in table



Id	Title	Author	Status
B001	harry potter	jk rowling	Available



ID	Name	Address	Contact No

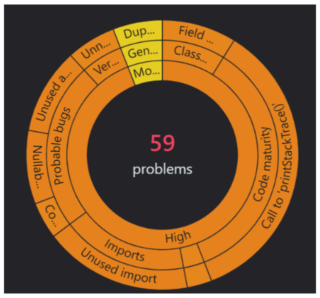
No content in table

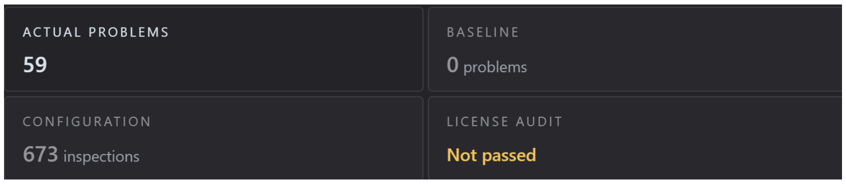
Github Usage :

- Link: https://github.com/laalateeq22/SWE303_Project_Layan_Klea.git
- We have used github in obtaining our project
- Communication of the updated versions of the code
    - This was particularly useful when we had to fix the code, since we both found some solutions to share with the other person
- To work with Qodana we used it through linking it with the github repository

Qodana usages

Qodana Before:





ACTUAL PROBLEMS	BASELINE
59	0 problems
CONFIGURATION	LICENSE AUDIT
673 inspections	Not passed

We actually used Qodana after we got the project to work. Some of the errors showed us like unused imports of data that we haven't used since we removed it to suit our project better. We had difficulty making Qodana work, but eventually we did it.

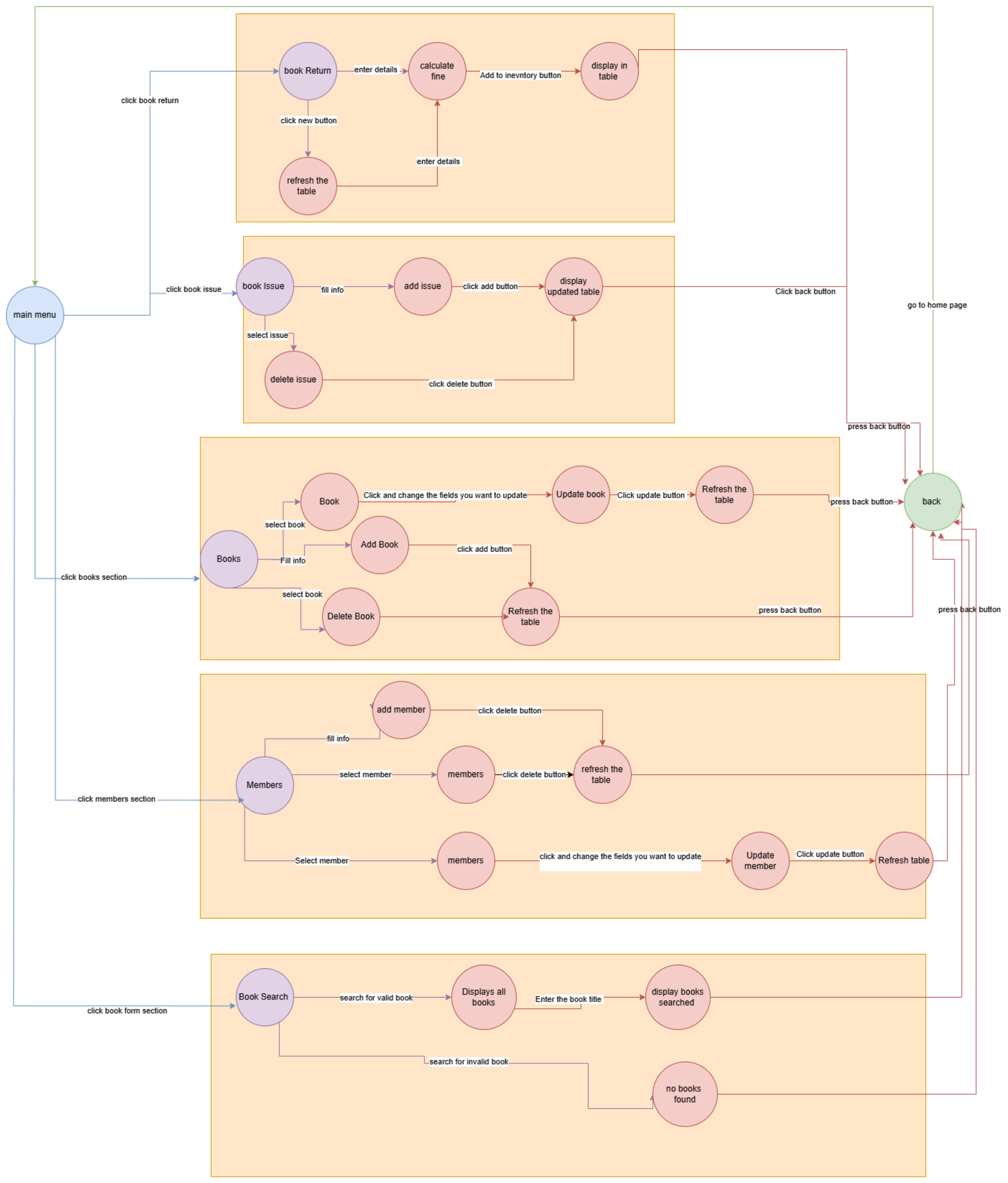
Qodana showed us 4 main types of errors

1. Duplicate Errors, which could not be fixed since the duplicates exists in different classes which perform similar functionalities therefore having duplicates
2. Saying that we should use robust logging instead of printing the stack trace
3. Other general issues with the code like return statements that are there to fill the catch block or saying that a variable is always declared as null.
4. And many unused import problems.

Struggles we faced

- The Java classes were all using JavaFX, therefore we couldn't perform the testing properly and we knew that this was an issue very late during the project since we tried to implement the coverage with fx, our current goal is to have at least some classes that perform the functions separately than with the fx.
    - This is currently a work in progress considering there are a lot of dependencies and this is not our code so the logic is a bit difficult to implement right away
- We struggled in getting the project to work with our computers since it was considered a little outdated therefore we had to refactor the code to suit our needs by removing the FXML details in the project, and by using an older version of JDK{17.0} instead of 23 which worked best on our code
- The selection of the project was quite difficult when we first started looking for projects that met the requirements and were still up to date but eventually, we found one that was good to refactor.
- With the use of Mockito libraries, our program couldn't recognize some of the text boxes which was quite a struggle for us.
- From the start, we had problems making the project work with the Scene builder, but with a thorough search,g we found the right library and we made it run.
- Then the Qodana part was the one that took us many days to solve. First Qodana did not scan our project at all, like it showed 0 errors because it didn`t scan any files. This was a process of trial and error.
- We tried working with the original pushed files, but it turned out to be almost impossible, so we removed all the files and we pushed the project folder once again.
- This was almost like a new start for us, because maybe the project folder was now structured differently, but Qodana Finally worked.
- Then, the most difficult part for both of us was making the test classes work, as the Mockito didn't work. While trying to fix that, we sometimes encountered different problems.
- But in the end, we made it work.
- Problems with installing Mockito, it wouldn't install properly

State Machine Diagram



Layan Abdulaziz Alateeq Part: BookFormController

Functions:

btn_new(): this is the function to execute the add new button in the fx part of the code

ValidateInputB(): it checks if the book entries are in the correct format for the title and author and checks if the ID is not empty

getAllBooks(): function to get all the books in the database and to return them in a list if type BookTM

btn_add(): function to deal with the fx functionality of the add/update button, it calls the addBookToDatabese and updateBookToDatabese functions to perform the request.

addBookToDatabese(): first calls the validate function to check if the book is in the correct format then it adds it to the database

updateBookToDatabese(): first finds the book, checks if the update is valid, then updates the book in the database

btn_dlt(): deals with deleting the FX functionality of the button delete and class the deleteBook() function to delete the book from the database

deleteBook(): a function that finds the book in the database and then deletes it

refreshTable(): a function that refreshes the table after an action is performed, this is executed in the fx dealing functions

The rest are either helper functions or deal with the animation

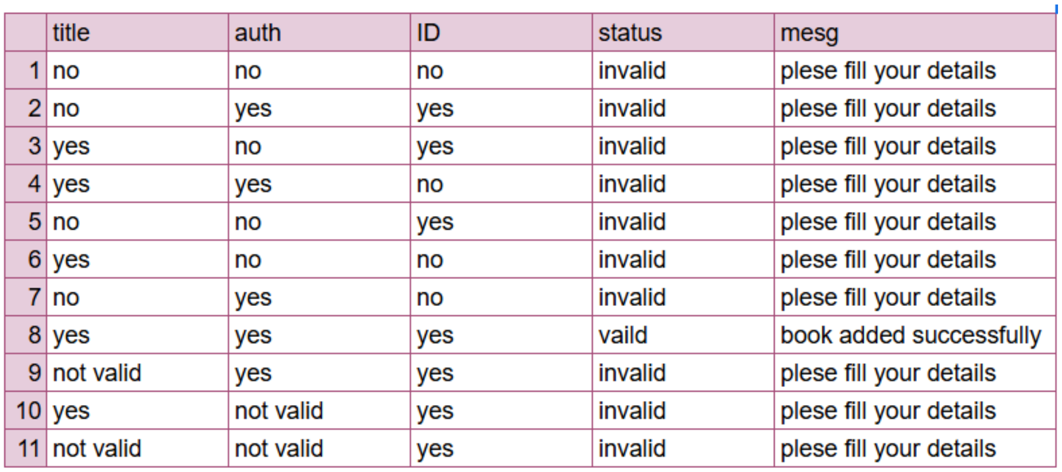
Testing Analysis {BookFormControllerTest}

testAddMultiple(): is for the boundary value testing to test if there is an error that could occur when entering a different number of books at the same time

TestAddBookWithMC_DCTest(): tests the selected cases of valid and invalid inputs ot test the function based on this table:

	ID	Title	Aurhor	Message
All Valid	MC1	title one	Author one	book added successfully
ID missing	MC2	title one	Author one	plese fill your details
Author Empty		title one		plese fill your details
Title Empty	MC4		Author one	plese fill your details

TestAddBookWithDecisionTable(): tests when different inputs are entered for valid and invalid cases based on this table:



	title	auth	ID	status	mesg
1	no	no	no	invalid	plese fill your details
2	no	yes	yes	invalid	plese fill your details
3	yes	no	yes	invalid	plese fill your details
4	yes	yes	no	invalid	plese fill your details
5	no	no	yes	invalid	plese fill your details
6	yes	no	no	invalid	plese fill your details
7	no	yes	no	invalid	plese fill your details
8	yes	yes	yes	vaild	book added successfully
9	not valid	yes	yes	invalid	plese fill your details
10	yes	not valid	yes	invalid	plese fill your details
11	not valid	not valid	yes	invalid	plese fill your details

Unit Testing {BookFormUnitTest}

setup(): calls the mokitoAnnotations.openMocks(this); omock the connection class using mokito

testAddAndRetriveBook(): tests adding a valid book to the database and checks if the book is added successfully to the database

testUpdateAndRetriveBook(): First, it adds a book to the database, then updates it with a valid update, checks if the update fits the format of the book structure, alters the data in the database, and checks if it's completed correctly in the database

testAddMultiplBooksAndRetrive(): adds different amounts of books, then checks it added to the database correctly.

Integration Testing
BookFormControllerTest and BookFormIntgTest classes

testAddAndRetriveBook(): tests adding a valid book to the database and checks if the book is added successfully to the database

testUpdateAndRetriveBook(): First, it adds a book to the database, then updates it with a valid update, checks if the update fits the format of the book structure, alters the data in the database, and checks if it's completed correctly in the database.

testAddAndDelete(): adds a book then checks if it is added to the database successfully, then we select the book and delete it then we check if it is deleted in the database

testAddMultiplBooksAndRetrive(): adds different amounts of books, then checks it added to the database correctly

testUpdateFunction(): adds a book, then changes it with different invalid and valid situations, and checks if it updates or rejects successfully

testDeleteBook(): adds book then it tries to select a book that doesn't exist, exist or is invalid then it tries to delete it and return if the book is deleted correctly or is facing some issues

System Testing {BookFormSystemTest}

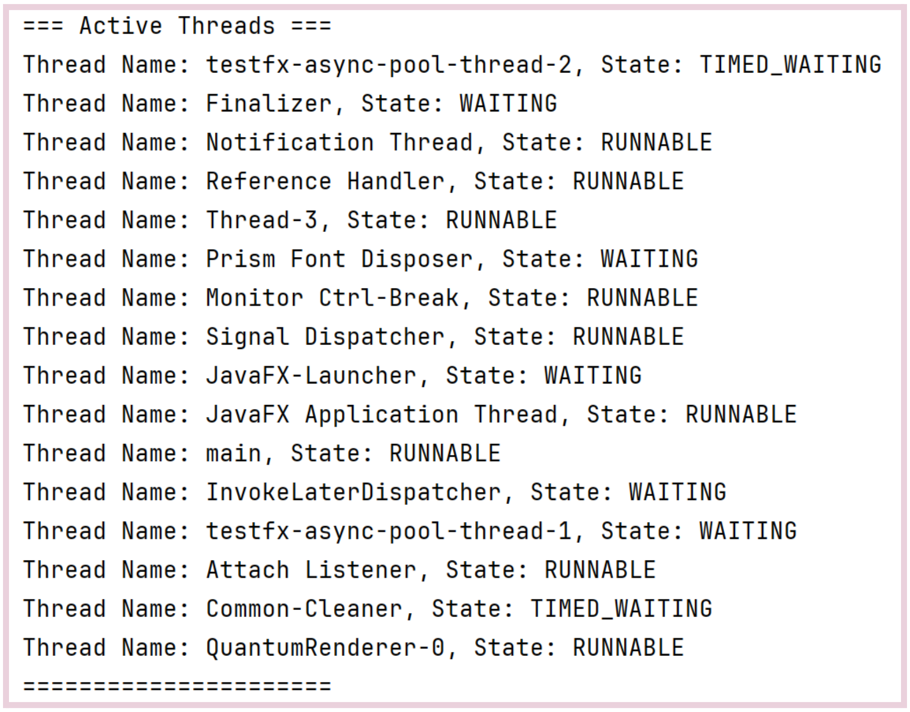
testAddBookThroughUI():

Path:

1. Click Books
2. Click new button
3. Click on the book ID button, delete the data, then write the new ID
4. Click on the book title and write the new title

5. Click on the book author and write the new author
6. Click on the add button
7. Check if it's found in the table
8. Clicks ok on the Alert

Threads:



```
=== Active Threads ===
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: QuantumRenderer-0, State: RUNNABLE
======================
```

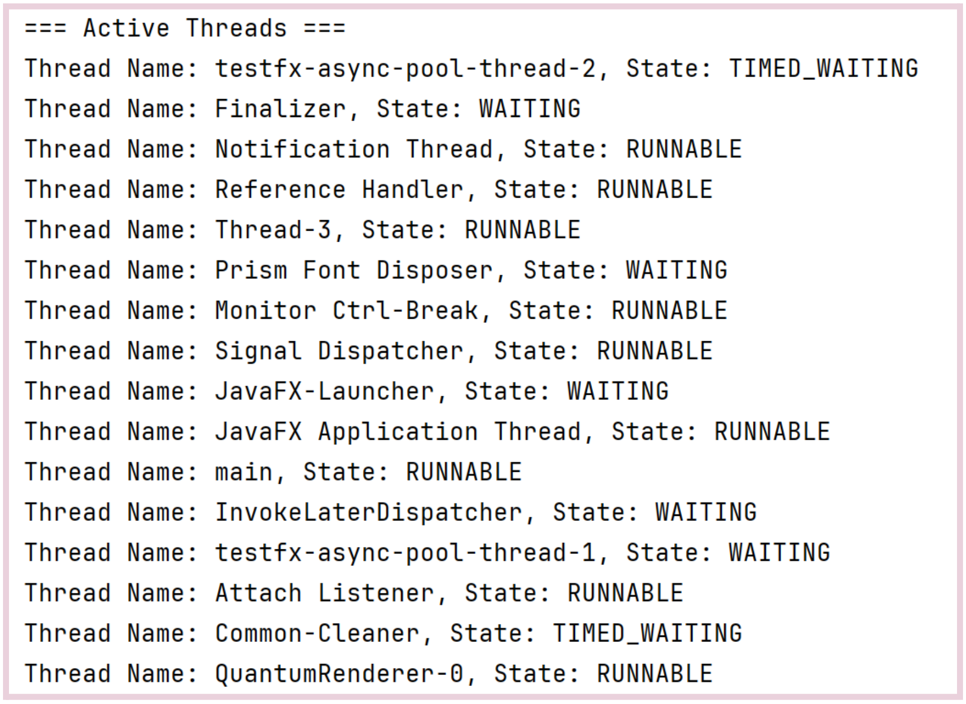
testUpdateBookThroughUI():

Path:

1. Click BooksClick Books
2. Click new button
3. Click on the book ID button, delete the data, then write the new ID
4. Click on the book title and write the new title
5. Click on the book author and write the new author
6. Click on the add button
7. Check if it's found in the table
8. Clicks ok on the Alert
9. Search for the book in the Table
10. Click on the found book
11. Update book title
12. Update book author

13. Check if updated correctly
14. Click okin alert shown

Threads:



```
=== Active Threads ===
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: QuantumRenderer-0, State: RUNNABLE
```

testDeleteThroughUI():

Path:

9. Click Books
10. Click new button
11. Click on the book ID button, delete the data, then write the new ID
12. Click on the book title and write the new title
13. Click on the book author and write the new author
14. Click on the add button
15. Check if it's found in the table
16. Clicks ok on the Alert
17. Search for the book
18. Click on the book
19. Click delete button
20. Check if it has been deleted
21. Click ok on the alert

Threads:

```
=== Active Threads ===
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: QuantumRenderer-0, State: RUNNABLE
======================
```

MemberFormController

Functions:

btn_Add(): deals with the FX part of the code for when when you want to add or update a member, into the database. It calls insertMember() to add the member to the database, and updateMember() to update a previous member.

insertMember(): checks if the member is valid then it inserts a member into the database and checks if it's entered correctly

updateMember(): First select a member then you update then you check if the updated version is a valid input using the function ValidateInput(), then you update it to the database, then you check if it is updated correctly.

btn_dlt(): deals with the FX functionality of the delete button and calls the delete member to remove the member from the database

deleteMember(): searches for the book in the database, then removes it from the database, then checks if the book is deleted from the database

ValidateInput(): it checks if the format of member details(name, address, phone number are valid, and when the ID and other variables are empty)

GetAllMembers(): a function that returns a list of type MemberTM to obtain all the members in the database for it to be used in other functions

loadAllMembers(): this function is used in the FX functionality, one of the old functions that used to create an observable list for the members, but since I changed the code I don't use it.it primarily exists because there is a functionality in the code that is dependent on it

generateNewID(): this function is used for generating a random ID. This is used when the new button is clicked to help the user with the next available ID if they do not have a unique one

The rest are either helper functions or deal with the animation

Testing Analysis

testAddMembersWithDecisionTable(): tests when different inputs are entered for valid and invalid cases based on this table:

1. When they are all empty
2. ValidID , invalid name
3. Valid ID, invalid Address
4. Valid ID, Invalid contact
5. All valid fields
6. Invalid ID

testAddMultipleMembers(): is for the boundary value testing to test if there is an error that could occur when entering a different number of members at the same time

Integration Testing

MemberFormControllerTest and MemberFormIntgTest classes

setup(): creates the database connection, initializes the variables it is performed before every functions/ test case in the class

initToolKit(): I tried using this to fix the fx in the junit testing but i still hasnt been a success since errors still arise which from what I understood was from the fx part. This is execute once before any other function in the test class

tearDown(): is executed after every function to remove any data they have added after the function used to tear down the connection that we have with the database

addAndRetriveMember(): tests adding a valid member to the database and checks if the book is added successfully to the database

testUpdateAndRetriveMember(): First, it adds a memberto the database, then updates it with a valid update, checks if the update fits the format of the member structure, alters the data in the database, and checks if it's completed correctly in the database.

testAddAndDeleteMember(): adds a Member then checks if it is added to the database successfully, then we select the member by searching for it and delete it then we check if it is deleted in the database

testAddMultipleMebrAndRetriveAll(); adds different amounts of Members, then checks it added to the database correctly

testDeleteMember(): adds member then it tries to select a book that doesn't exist, exist or is invalid then it tries to delete it and return if the member is deleted correctly or is facing some issues

testUpdateMember(): adds a book checks if its added correctly, then changes it with different invalid and valid situations, and checks if it updates or rejects successfully

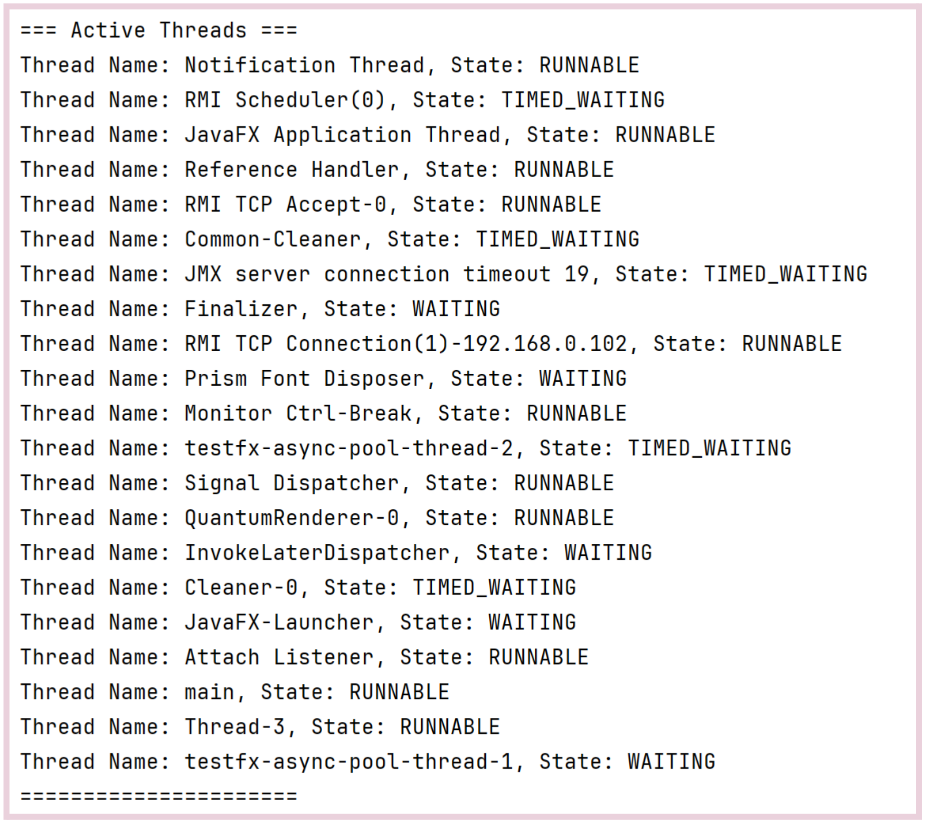
System Testing{MemberFormSystemTest}

testAddMemberThroughUI():

Path

1. Click on members
2. Click new button
3. Click on ID and delete the data,then write a name
4. Click address, then add address
5. Click contact, then add new contact
6. Click on button add
7. Make ure its added succesfully
8. Click ok on the alert shown

Threads



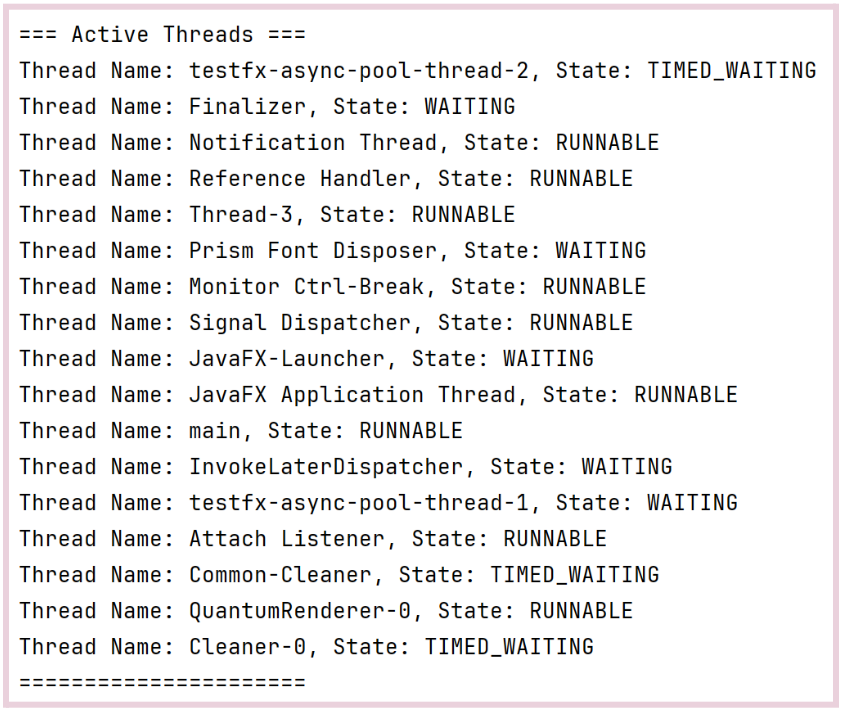
```
=== Active Threads ===
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: RMI Scheduler(0), State: TIMED_WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: RMI TCP Accept-0, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: JMX server connection timeout 19, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: RMI TCP Connection(1)-192.168.0.102, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: QuantumRenderer-0, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: Cleaner-0, State: TIMED_WAITING
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: testfx-async-pool-thread-1, State: WAITING
======================
```

testUpdateMemberThroughUI():

Path

1. Click on members
2. Click new button
3. Click on ID and delete the data,then write a name
4. Click address, then add address
5. Click contact, then add new contact
6. Click on button add
7. Make ure its added succesfully
8. Click ok on the alert shown
9. Search through table for the member
10. Click the member
11. Update the member details
12. Click update button
13. Check if updated successfully
14. Click ok for the alert

Threads



```
=== Active Threads ===
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: QuantumRenderer-0, State: RUNNABLE
Thread Name: Cleaner-0, State: TIMED_WAITING
======================
```

testDeleteMemberThroughUI():

Path

Click on members

1. Click new button
2. Click on ID and delete the data,then write a name
3. Click address, then add address
4. Click contact, then add new contact
5. Click on button add
6. Make ure its added succesfully
7. Click ok on the alert shown
8. Search for the added member
9. Lcickon the member
10. Click on the delete button

Threads

```
=== Active Threads ===
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Finalizer, State: WAITING
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Prism Font Disposer, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: QuantumRenderer-0, State: RUNNABLE
======================
```

HomeFormController

Functions:

navigate(): is a unction that deals with the fx part of seeing which button the userwill chose, anddirectsit to the appropriate class

The other function is used for the animation of the pages

System Testing {HomeFormSystemTest}

I split into 2 functions because the program was overwhelmed when i put in in one functions

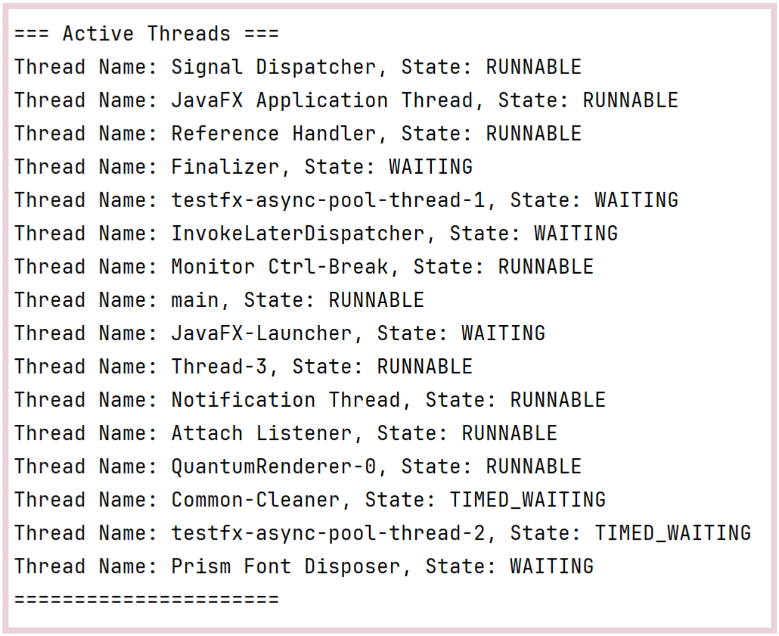
homeSystemTest1():

Path

1. Click on members
2. Click back
3. Click on books
4. Click back

5. Click on issue
6. Click back

Threads



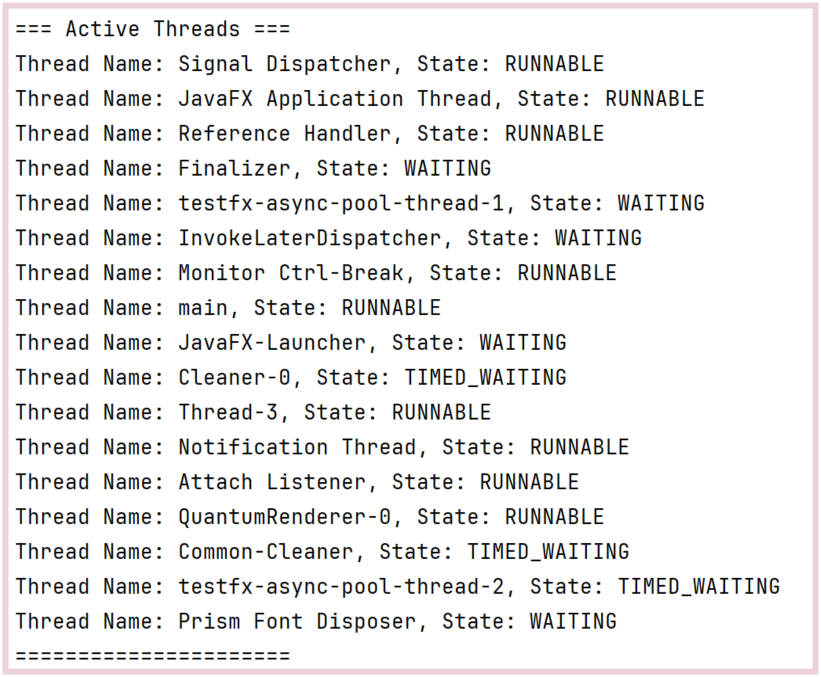
```
=== Active Threads ===
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Finalizer, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: QuantumRenderer-0, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Prism Font Disposer, State: WAITING
======================
```

homeSystemTest2():

Path

1. Click return
2. Click back
3. Click search
4. Click Back

Threads



```
=== Active Threads ===
Thread Name: Signal Dispatcher, State: RUNNABLE
Thread Name: JavaFX Application Thread, State: RUNNABLE
Thread Name: Reference Handler, State: RUNNABLE
Thread Name: Finalizer, State: WAITING
Thread Name: testfx-async-pool-thread-1, State: WAITING
Thread Name: InvokeLaterDispatcher, State: WAITING
Thread Name: Monitor Ctrl-Break, State: RUNNABLE
Thread Name: main, State: RUNNABLE
Thread Name: JavaFX-Launcher, State: WAITING
Thread Name: Cleaner-0, State: TIMED_WAITING
Thread Name: Thread-3, State: RUNNABLE
Thread Name: Notification Thread, State: RUNNABLE
Thread Name: Attach Listener, State: RUNNABLE
Thread Name: QuantumRenderer-0, State: RUNNABLE
Thread Name: Common-Cleaner, State: TIMED_WAITING
Thread Name: testfx-async-pool-thread-2, State: TIMED_WAITING
Thread Name: Prism Font Disposer, State: WAITING
======================
```

BookIssueFormController Class {KLEA}
This is the class I have chosen to perform the Class Evaluation Testing and Code Coverage Testing (Branch Coverage.)

⇒ ***ClassEvaluationTesting***

The class BookIssueFormControllerTest tests the new_action method of the BookIssueFormController class. The method resettes UI fields, generates a new issueID and sets the new ID. The test class goes over all the scenarios:
`newAction_EmptyTable`→It verifies that the method initializes the ID correctly on an empty table scenario. It should be I001.
`newAction_WithExistingSequentialRecords`→Determines the next sequential ID by finding the highest existing ID and incrementing it. After the code runs it checks if the ID is incremented to the right value..
`newAction_WithNonSequentialRecords`→ Populates the issuetb table with non-sequential IDs and shows the robustness of the method in handling gaps in ID sequences.
`newAction_DatabaseConnectionError`→ Simulates a database connection failure by closing the connection before invoking new_action and validates error handling in the absence of database connectivity.
`newAction_WithLargeNumberOfRecords`→ Populates the issuetb table with up to 1000 records and makes sure the method generates the next ID efficiently, even with a large dataset.

Branch Coverage Testing

The two test classes may look alike, because they are similar in functionality, but they differ in comprehension and their coverage of edge cases. Both classes test in their two first test cases for scenarios when the database is empty (an empty table), and for sequential IDs. Both test classes test in their third test case for gaps in ID, but the `BookIssueFormControllerTest` assumes the method generates the next available ID ignoring gaps (`I004` for gaps after `I001` and `I003`). The `BranchCoverageTest` correctly ensures the smallest missing value is chosen (`I002` for a gap between `I001` and `I003`). Both test classes test the code functionality with large datasets.The BranchCoverageTest Class focuses on testing edge cases.

`NewAction_DatabaseError_NoTable`→ Ensures that when the table is missing, new_action throws an exception and the error message references the missing table.
`NewAction_NullDatabaseConnection`→ Does the same thing as the above test case, just instead of no table it references a null database connection.
`NewAction_NonNumericIDValues`→ Verifies that malformed IDs are ignored when

generating the next ID, ensuring that valid numeric IDs (`I002`) are still correctly computed.
`NewAction_MissingColumn`→ Simulates a schema change where the `issueId` column is missing and ensures that the method throws an exception, with an error message indicating the missing column.

BookReturnFormController Class

Boundary Value Testing

The test class BookReturnFormControllerTest tests the functionality related to fine calculation for overdue book returns at the BookReturnFormController class.
`FineCalculation_AtExactBoundary_NoFine`→ Tests no fine on the 14th day (last day of grace period).
`FineCalculation_JustOverBoundary`→ Tests fine of 15 on the 15th day (start of fine period).
`testFineCalculation_JustBelowBoundary`→ Confirms no fine on the 13th day (within grace period).
`FineCalculation_AtStartDate`→ Ensures no fine for same-day return.
`FineCalculation_LongAfterBoundary`→ Validates fine accumulation for extended overdue returns.

BookIssueFormController

1. Unit Tests:

testInitialize()

Expected Output: The `initialize()` method should execute successfully without throwing errors, ensuring the controller is correctly initialized.

Result: Verifies that the `initialize()` method prepares the database connection and sets up UI components.

testNewActionGeneratesNewId()

Expected Output: Clicking the "New" button generates a valid issue ID.

Result: Ensures that the "New" button triggers the generation of a valid issue ID.

Reason: To verify that a new, valid issue ID is generated when the "New" button is clicked.

testAddActionWithValidData()

Expected Output: A valid entry is added to the database, and the `TableView` is updated.

Result: Ensures that valid input data triggers the appropriate database insert and updates the UI.

testAddActionWithInvalidData()

Expected Output: No entry is added if the form data is invalid, and no database action occurs.

Result: Verifies that invalid data prevents form submission and database interaction.

testDeleteActionWithValidData()

Expected Output: A selected issue is deleted from the database, and the TableView is updated.

Result: Verifies that valid delete actions are executed and the UI is updated.

testDeleteActionWithoutSelection()

Expected Output: No delete action occurs if no row is selected, and an alert is displayed.

Result: Verifies that the delete action only triggers when a row is selected.

testAddActionWithInvalidIssueDate()

Expected Output: No action occurs when the issue date is invalid, and the system handles it gracefully.

Result: Verifies that invalid issue dates prevent form submission and database changes.

testAddActionWithEmptyFields()

Expected Output: No database interaction occurs when the form fields are empty.

Result: Ensures that empty fields prevent form submission.

Test handling invalid member IDs (t**estMemberIdSelectionHandlesInvalidId())**: Ensures the controller handles invalid member IDs properly without updating the name field.

Expected Output: If an invalid book ID is selected, the book title field should remain empty.

Result: The test verifies that the controller handles invalid book IDs correctly without updating the txt_title field.

Test handling invalid book IDs (**testBookIdSelectionHandlesInvalidId()**): Ensures the controller handles invalid book IDs correctly by leaving the title field empty.

Test for unavailable books (**testBookIdSelectionWhenBookUnavailable()**): Ensures that if a book is unavailable, the system displays an appropriate message and doesn't proceed with the book issue.

Test for invalid issue date format handling (**testIssueDateFormatHandling()**): Verifies that invalid issue dates are handled without throwing errors.

Test for valid member and book selections updating fields (**testValidMemberAndBookIdShouldPopulateFields()**): Verifies that when valid member and book IDs are selected, their corresponding fields are updated.

2.Integration Tests

`testLookupButtons()`:
Expected Output: Verify that buttons (btnNew, btnAdd, btnDelete) exist in the UI.
Result: Ensures that the required buttons are present in the UI.

`testNewActionGeneratesNewIssueId()`:
Simulates clicking the "New" button and checks that a new issue ID is generated in the `txt_issid` field.
Expected Output: Clicking the "New" button should generate a valid issue ID in the txt_issid field.
Result: Verifies that the "New" action triggers the correct generation of an issue ID.

`testMemberSelectionUpdatesNameField()`:

Simulates selecting a member and checks that the `txt_name` field is populated with the member's name.
Expected Output: Selecting a member from the ComboBox should populate the txt_name field.
Result: Verifies the integration between the ComboBox and the txt_name field.

`testBookSelectionUpdatesTitleField()`:

Simulates selecting a book and verifies that the `txt_title` field is populated with the book's title.
Expected Output: Selecting a book from the ComboBox should populate the txt_title field.
Result: Verifies the integration between the ComboBox and the txt_title field.

`testInvalidMemberIdSelection()`:

Simulates selecting an invalid member and ensures that the `txt_name` field is not updated.

Expected Output: An invalid member selection should leave the txt_name field unchanged.

Result: Ensures that invalid member IDs are handled correctly

`testNewAction()`:

Simulates clicking the "New" button with an `ActionEvent` and verifies a new issue ID is generated.

Expected Output: Clicking the "New" button should generate a valid issue ID.

Result: Verifies that the issue ID generation works as expected when the button is clicked.

`testAddAction()`:

Simulates adding a new issue by selecting a member, book, and date, then verifies the `TableView` is updated with the new issue.

Expected Output: After selecting a member, book, and entering a date, the new issue should be added to the TableView.

Result: Verifies that the "Add" action works correctly and updates the UI.

3.System Test:

Test "Add" action (`testAddAction()`):

Simulates the process of adding a new book issue and checks if the table is updated correctly after adding.

Expected Output: A new issue should be added, and the TableView should be updated accordingly.

Result: Simulates adding an issue and ensures that the table updates after the action

Test "Delete" action (`testDeleteAction()`):

Simulates deleting an issue and verifies that the item is removed from the table view.

Expected Output: A selected issue should be deleted from the TableView.

Result: Ensures the delete action updates the UI and removes the selected issue.

Test navigating to the Book Issue Form (`testNavigateToBookIssueForm()`):

Verifies that the navigation from the home form to the Book Issue form works as expected.

Expected Output: Navigation from the home form to the Book Issue form should work smoothly.

Result: Ensures that the navigation between views is seamless.

Test searching for an issue (`testSearchIssueFunctionality()`):

Simulates searching for an issue by ID and checks if the table view updates with the search results.

Expected Output: Searching for an issue by ID should update the TableView with the matching results.

Result: Verifies that the search functionality works correctly and displays results.

Test deleting an issue (`testDeleteIssue()`):

Ensures that selecting an issue in the table and clicking delete removes the issue from the table.

<u>**Threads**</u>:

1. Starts from the Home Screen
   - The `start()` method loads the `HomeFormView.fxml` file.
   - This initializes the main menu of the Library Management System.
2. Navigates to the Book Issue Form
   - The test clicks on the "Issue Book" button (`#issue`), which navigates to the Book Issue form.
3. Loads the Book Issue Screen (`BookIssueForm.fxml`)
   - The application loads `BookIssueForm.fxml`, initializing the issue book table and input fields.
4. Tests Book Issue Functionality
   - Verifies that the table is visible (`#bk_ssue_tbl`).
   - Performs a search operation (`testSearchIssueFunctionality`).
   - Tests deleting an issue (`testDeleteIssue`).

BookReturnFormController

1.Unit Tests:

`testGetIssueDate()`:

Tests the `getIssueDate()` method by mocking the database interaction to verify that the correct issue date is retrieved for a given issue ID.

Expected Output: Correct issue date is returned based on the given issue ID.

Result: Verifies that the issue date is correctly retrieved from the database.

`testCalculateFine()`:

Tests the static `calculateFine()` method to ensure it calculates the fine correctly based on the issued date and returned date.

Expected Output: Fine is calculated based on the difference between the issued and returned dates.

Result: Verifies that the fine calculation logic works as expected.

`testAddReturnRecordSuccess()`:

Verifies that the `addReturnRecord()` method successfully inserts a return record into the database when given valid input.

Expected Output: A return record is successfully inserted into the database.

Result: Ensures that return records are added correctly when valid data is provided.

`testAddReturnRecordFailure()`:

Verifies that the `addReturnRecord()` method returns `false` when the database insert fails (i.e., no rows are affected).

Expected Output: Return record is not added if the database insert fails.

Result: Verifies that the controller correctly handles failed insert attempts.

`testUpdateBookStatus()`:

Tests the `updateBookStatus()` method to ensure the book's status is updated in the database when a return record is successfully added.

`testCalculateFine_NoLateReturn()`:

Verifies that the `calculateFine()` method returns `0.0f` for a return that is not late.

`testCalculateFine_LateReturn()`:

Tests that the `calculateFine()` method correctly calculates a fine for late returns.

`testInitializeDatabaseConnection()`:

Verifies that the database connection is correctly initialized and is valid.

2.Integration Tests:

`testAddReturnRecord_Success()`:
Verifies that the `addReturnRecord()` method correctly interacts with the database and adds a return record when valid data is provided.
Expected Output: Valid return data is added to the database, and the UI is updated.
Result: Verifies that the add return record logic interacts with the database and updates the UI.

`testAddReturnRecord_Failure()`:

Verifies that the `addReturnRecord()` method handles failure correctly when the database insert operation doesn't affect any rows.

`testUpdateBookStatus()`:

Ensures that the `updateBookStatus()` method correctly updates the status of a book in the database after a return record is added.

Expected Output: The book's status is updated after the return is processed.

Result: Verifies that updating the book status works correctly after a return record is added.

`testComboBoxPopulation()`:

Verifies that the ComboBox (`cmb_issue_id`) is populated with data from the `issuetb` table and reflects any changes made to the database.

Expected Output: The ComboBox (cmb_issue_id) is populated with valid issue IDs from the database.

Result: Verifies that the ComboBox is populated correctly with available issue IDs.

`testLoadInitialData()`:

Verifies that the `loadInitialData()` method correctly loads return records from the `returndetail` table into the `rt_tbl` TableView and populates the ComboBox with issue IDs.

Expected Output: Return data is loaded from the database into the rt_tbl TableView.

Result: Verifies that the loadInitialData() method works and the data is displayed in the TableView.

<u>BookSearchFormController</u>

1.Unit Tests

`testSetupTableColumns()`:

Verifies that the columns for the TableView (`tbl_bk`) are set up correctly. It checks if the TableView has the expected number of columns and verifies the column titles.

Expected Output: The columns of the TableView should be properly set up.

Result: Ensures the columns (id, title, author, status) are added correctly to the TableView.

`testLoadAllBooks()`:

Tests the `loadAllBooks()` method which retrieves all books from the database. It ensures that after the books are loaded, the table is populated with data and is not empty.

Test Case: testLoadAllBooks()

Expected Output: All books are retrieved from the database and displayed in the TableView.

Result: Verifies that the books are successfully loaded from the database into the TableView.

2.Integration tests:

`testTablePopulation()`:
Simulates loading the data into the `TableView` after executing `loadAllBooks()`. It verifies that the data from the database is correctly inserted into the `TableView`.

Expected Output: Books retrieved from the database are correctly populated in the TableView.
Result: Ensures the TableView is populated with data from the database after executing loadAllBooks().

`testSearchFunctionality()`:
Verifies the search functionality by simulating text input into the `TextField` (`bk_sch`). It checks if the `TableView` is updated correctly based on the search input. The test ensures that books matching the search criteria are displayed in the table.
testSearchFunctionality()

Expected Output: Books matching the search query are displayed in the TableView.
Result: Verifies the search functionality works and updates the table view based on the search query.

`testNavigationToHomePage()`:
Verifies the navigation functionality by simulating a click on the back button (assumed to have the ID `img_bk`). The test ensures that when the user navigates back, the correct scene (Home Page) is displayed.

Expected Output: Clicking the back button navigates to the home page.
Result: Verifies that the user can successfully navigate back to the home page.

3. System Test

`testBookSearchAndReturnToMainPage()`:

Simulates navigating to the Book Search form by clicking on an element (e.g., `bk_search`).

Verifies that the Book Search form is loaded with the `TextField` for search and `TableView` for displaying books.

Simulates typing a search term (`B001`) into the search field and pressing "Enter". It verifies that the search results display books matching the search term.

Verifies that the results in the `TableView` contain books that match the search criteria.

Simulates returning to the main page by clicking on the back button (`img_bk`).

Verifies that the main page (Home Form) is displayed correctly.

Expected Output: Verifies that the user can search for a book, view the results, and return to the home page.

Result: Ensures that the search functionality and navigation are working end-to-end.

<u>Threads:</u>

1. JavaFX Thread → Load Home Screen (`HomeFormView.fxml`)
2. TestFX Background Thread → Click "Issue Book" (`clickOn("#issue")`)
3. JavaFX Thread → Load Book Issue Form (`BookIssueForm.fxml`)
4. TestFX Background Thread → Perform Search (`write("SS12345")`)
5. JavaFX Thread → Update TableView with search results
6. TestFX Background Thread → Select issue in TableView (`selectFirst()`)
7. TestFX Background Thread → Click "Delete" (`clickOn("#btnDelete")`)
8. JavaFX Thread → Remove issue from TableView