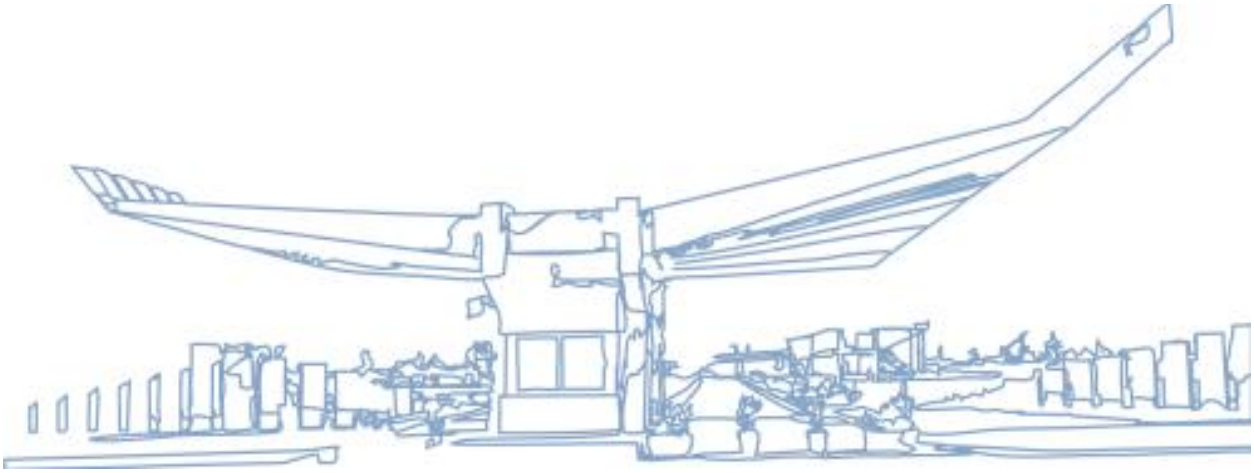
Department of Computer Engineering

Course Name: Software Testing and Quality Assurance
Course Code: SWE 303

Project Title:

BOOKSTORE MANAGEMENT SYSTEM SOFTWARE TESTING



Group Members:

1. Borian Llukaçaj
2. Engjëll Abazaj
3. Indrit Ferati

Introduction

The Bookstore Management Software is designed to streamline and enhance the operations of a library or bookstore by offering an efficient and user-friendly system for managing books, transactions, and staff. Built with a robust JavaFX graphical user interface and backed by a MySQL database, the software provides a comprehensive solution for handling inventory, sales, and employee records. It ensures that essential data such as book details, sales transactions, and stock levels are securely stored and easily accessible, allowing the bookstore to operate smoothly and effectively. The system is also built using an MCV (Model-View-Controller) architecture.

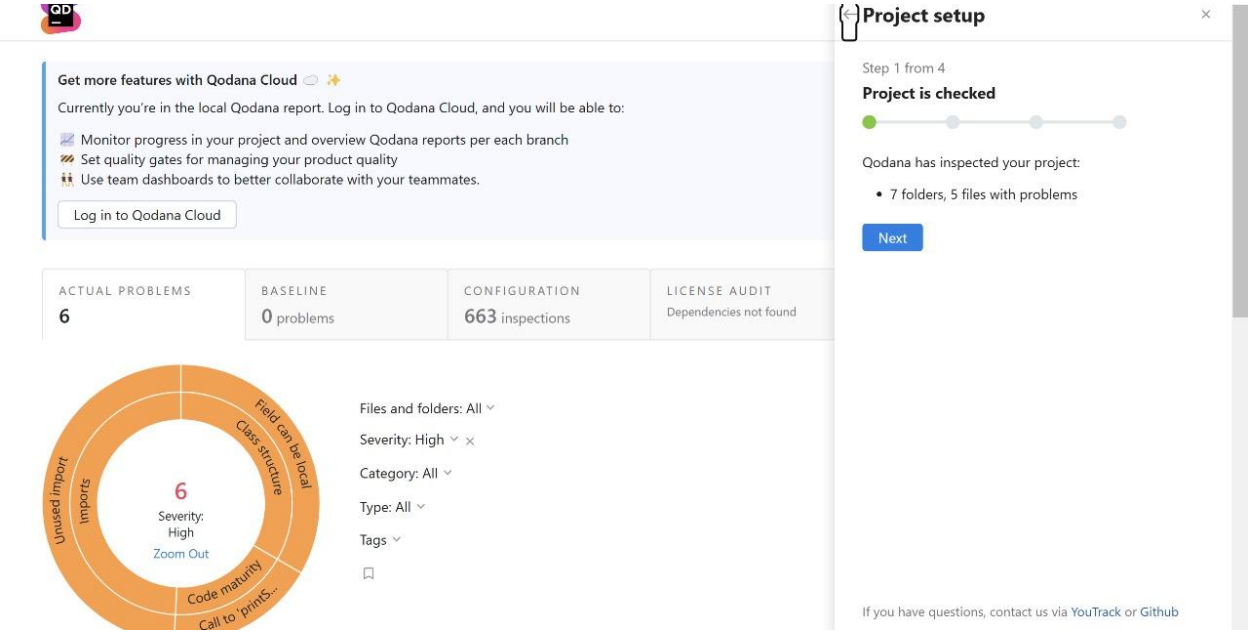
The aim of this project is to perform different kinds of software testing on this project, to ensure that everything works as intended and determine any problems that can be resolved in the future.

GitHub Repository: *https://github.com/Diti2604/bookstore-management-system*

Static Testing (Qodana)

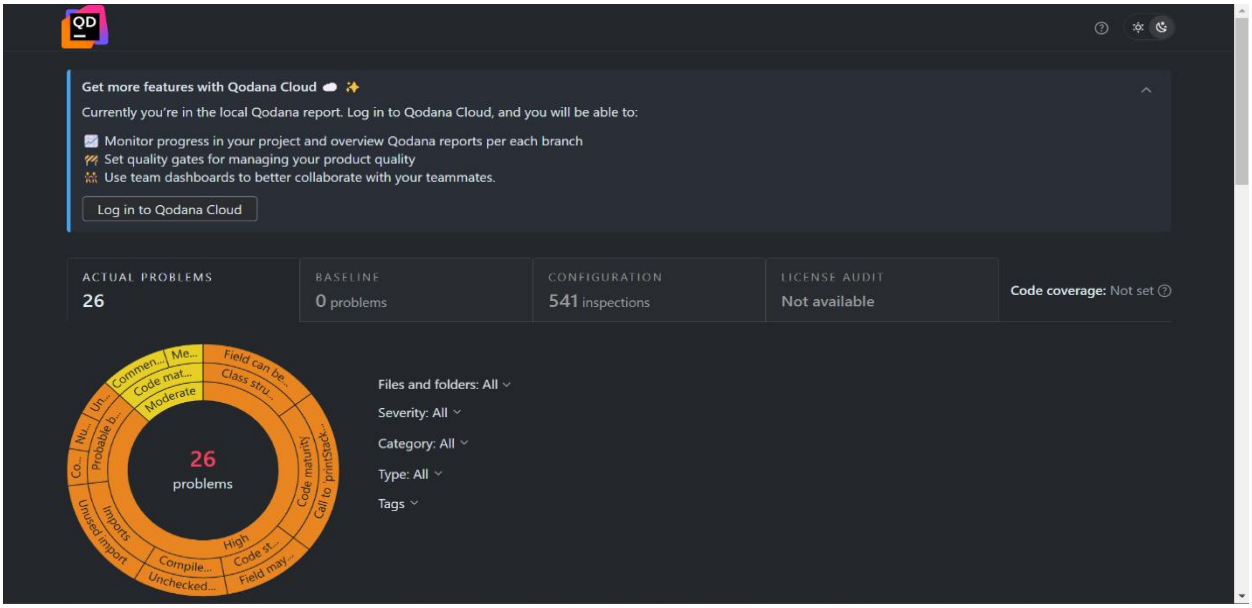
When the Qodana process in our project was done, we noticed that we had 62 problems. In order for all of us to contribute to static testing we divided our work according to the MVC model. Borjan was responsible for the Controller package classes, Engjëll for the Model package classes and Indrit was responsible for the View package classes. Below are shown the respective screenshots of Qodana tests and their explanations.

Model – Engjëll



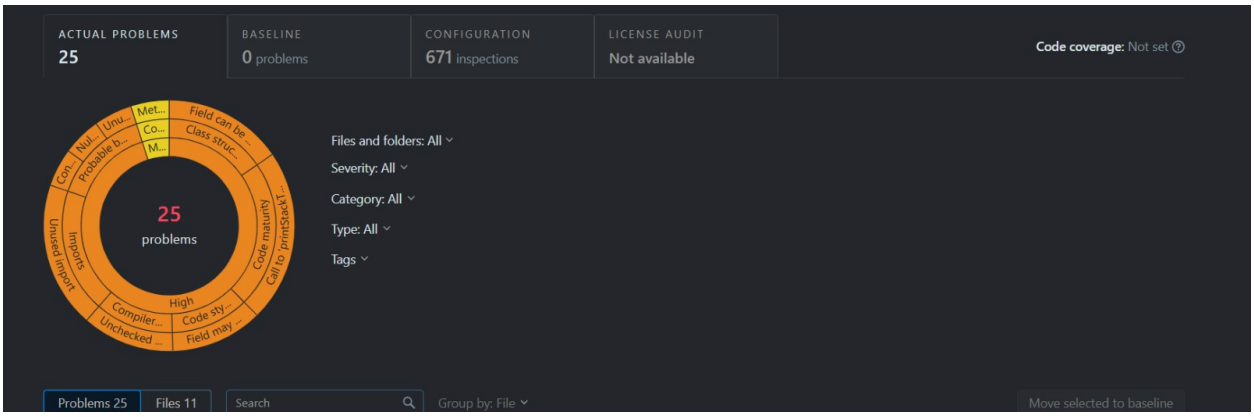
The Model package seemed to have the smallest number of errors in the code. The majority of the problems that we fixed were printStackTraces that needed to be substituted with meaningful messages, unused imports and unused variables, or variables that could have been declared locally as they were used only once inside a certain scope.

View – Indrit



The View part was responsible for 26 problems where the majority of it was to fix the printStackTraces() which required changing them with some meaningful messages. The other errors were unused imports and unused variables and changing some variables access modifier.

Controller – Borjan



The Controller package contained 25 problems which were of the same type as the other classes. Each issue was dealt with accordingly.

Testing Analysis

Borian

Equivalence Partitioning (T1-T2)

- **Reasoning:** These tests divide the input domain into equivalence classes, selecting representative values from each class. This is efficient as it reduces the number of test cases while providing good coverage. This method was chosen because it clearly divides the outputs it can give depending on the strings you give it similar to how BB->pass and FF-> fail example was given in class in this case we can have a valid input or invalid, we can also have different outputs depending on if role is manger or something else.
    - **T1:** Represents the valid input class for the calculateSalaryByRoleAndTimeframe method.
    - **T2:** Represents the invalid input class, ensuring the method handles unexpected or erroneous data appropriately.

Boundary Value Analysis (T6-T9)

- **Reasoning:** This technique focuses on testing values at the boundaries of input and output ranges. Errors often occur at these limits. I chose this method to test because it was one of the few methods where we could find boundaries to test in this case the boundary was the acceptable selling price as it couldn't be negative or higher than a large number .
    - **T6:** Tests the lower boundary of the acceptable selling price.
    - **T7:** Tests the upper boundary of the acceptable selling price.
    - **T8:** Tests values below the lower boundary of the acceptable selling price.
    - **T9:** Tests values above the upper boundary of the acceptable selling price.

Coverage (T3-T5)

- **Reasoning:** This technique aims to execute all possible paths through the code, ensuring that all conditions and branches are exercised. There wasn't a big reason why this method would be chosen over another one however I chose a method that has medium path complexity so there were some path variation and not just none.
    - **T3:** Tests the code path when a book with a null cover image is encountered.
    - **T4:** Tests the code path when a book with a valid cover image is found.
    - **T5:** Tests the code path when no book is found for the given ISBN.

Test Case ID	Method Tested	Reason	Input	Expected Output	Comments
T1	calculateSalaryByRoleAndTimeframe	To ensure correct salary computation for a valid role and timeframe	Role: "Manager", Timeframe: "daily"	Salary: 200.0	Verifies correct daily salary calculation
T2	calculateSalaryByRoleAndTimeframe	To validate exception handling for invalid inputs	Role: "Manager", Timeframe: "invalid"	Exception: `IllegalArgumentException`	Ensures robust input validation
T3	getBookByISBN	Validate retrieval of book details without cover image	ISBN: "123456789"	Book object with null cover image	Ensures proper handling of null cover images
T4	getBookByISBN	Validate retrieval of book details with cover image	ISBN: "987654321"	Book object with valid cover image	Verifies correct processing of Blob data for cover images
T5	getBookByISBN	Validate behavior when no book is found for the given ISBN	ISBN: "000000000"	Null book object	Ensures method handles non-existent records correctly
T6	saveBook	Validate saving a book at lower price boundary	Book with selling price 1.0	Successful save operation	Ensures method processes minimum valid price correctly
T7	saveBook	Validate saving a book at upper price boundary	Book with selling price 100000.0	Successful save operation	Ensures method processes maximum valid price correctly
T8	saveBook	Validate rejection of book below lower price boundary	Book with selling price -1.0	No operation performed	Ensures method prevents invalid prices below boundary
T9	saveBook	Validate rejection of book above upper price boundary	Book with selling price 100001.0	No operation performed	Ensures method prevents invalid prices above boundary

Engjëll

Boundary Value Testing

calculateTotalBillCostWithTaxByTimeframe

I decided to perform BVT on this method since BVT focuses on testing the edge cases of input and this particular method offers a range of input values for the timeframe. The boundary conditions serve as test cases and they are represented in the table below.

Test Case	Method Tested	Reason	Input Value	Expected Output	Comments
Daily – Lower Bound	calculateTotalBillCostWithTaxByTimeframe("daily")	To test the lowest boundary (start of the day)	Today's Date	20.0	Making sure only 1 bill is considered
Daily – Upper Bound	calculateTotalBillCostWithTaxByTimeframe("daily")	To test the upper boundary (end of the day)	Today's Date	60.0	Making sure the calculation for multiple bills is correct
Weekly – Lower Bound	calculateTotalBillCostWithTaxByTimeframe("weekly")	To test the start of the week boundary	Start of the week	20.0	Making sure only the bills for one week are calculated
Weekly – Upper Bound	calculateTotalBillCostWithTaxByTimeframe("weekly")	To test the end of the week boundary	End of the week	60.0	Making sure bills for multiple weeks are summed
Monthly – Lower Bound	calculateTotalBillCostWithTaxByTimeframe("monthly")	To test the start of the month boundary	Start of the month	20.0	Making sure bills for the entire month are summed
Monthly – Upper Bound	calculateTotalBillCostWithTaxByTimeframe("monthly")	To test the end of the month boundary	End of the month	60.0	Making sure bills for the entire month are summed

Yearly – Lower Bound	calculateTotalBillCostWithTaxByTimeframe("yearly")	To test the start of the year boundary	Start of the year	20.0	Making sure bills for the entire year are summed
Yearly – Upper Bound	calculateTotalBillCostWithTaxByTimeframe("yearly")	To test the end of the year boundary	End of the year	60.0	Ensure bills for the entire year are summed

Class Evaluation Testing

calculateTotalBillCostWithTaxByTimeframe

This method is also a good candidate for class evaluation testing, because we can also test the interior of the test cases together with the edge cases.

Test Case	Reason	Input Value	Expected Output	Comments
Daily Timeframe with Valid ResultSet	Ensures the correct calculation of tax amount for daily timeframe with valid data	Today's Date	20.0	Checks if only one day of bills is considered
Weekly Timeframe with Multiple Records	Verifies that the method correctly handles multiple weekly bills and sums them properly	Start of Week, End of Week	80.0	Ensures correct summation of weekly bills
Monthly Timeframe with Empty ResultSet	Ensures the method correctly handles an empty monthly timeframe	Start of Month, End of Month	0.0	Verifies no bills return a zero tax amount
Yearly Timeframe with Negative Bill	Verifies handling of negative bill values	Start of Year, End of Year	-40.0	Ensures proper handling of negative values
Invalid Timeframe Throws Exception	Tests that invalid timeframes trigger an IllegalArgumentException	"hourly"	Exception	Ensures proper exception handling for invalid input
Zero Total Bill Amount	Ensures the method returns 0 when total bill amounts are zero	Today's Date	0.0	Verifies correct tax calculation for zero bill amounts

Code Coverage Testing

calculateTotalBillCostWithTaxByTimeframe

I am using the same method again, because it has multiple conditional statements that can help me do coverage testing.

Test Case	Type of Coverage	Reason	Input Value	Expected Output	Comments
Statement Coverage - Valid Daily Timeframe	Statement Coverage	Ensures the correct execution of all statements in the calculateTotalBillCostWithTaxByTimeframe("daily") method	"daily"	20.0	Verifies the tax calculation for daily timeframe
Branch Coverage - Invalid Timeframe	Branch Coverage	Tests the behavior of the method when given an invalid timeframe, ensuring that the correct exception is thrown	"invalid_timeframe"	Exception	Verifies proper exception handling for invalid input
Condition Coverage - Weekly Multiple Records	Condition Coverage	Tests the logic that handles multiple records for the weekly timeframe. Ensures all conditions in the code are covered	"weekly"	80.0	Validates summation of weekly bill records
MCDC Coverage (Monthly)	MCDC Coverage	Ensures that all combinations of conditions are tested for the monthly timeframe, covering all scenarios.	"monthly"	80.0	Validates different scenarios of monthly bill records.

Indrit

Boundary Value Testing

countBooksByPrice()

This method was chosen to do boundary value testing because it takes two arguments (int price, String sortBy) one of them being a numerical input and the other is used to clarify for which type of sorting is the price used for. The output of this method is a number of books which are the less than or equal to the Price given in the method parameter. Since the input and output is numerical and there is a range of numerical values that can be used, I chose this method to do BVT(Robust). The nominal value in the testing method was the average of the prices of all books.

Test Case	Method Tested	Reason	Input Value	Expected Output	Comments
Price at Lower Boundary	countBooksByPrice	To test lowest acceptable price	price = 0	book_count = 0	Verifies books priced at 0.
Price at Upper Boundary	countBooksByPrice	To test highest acceptable price	price = 1,000,000	book_count = 0	Ensures large prices are handled.
Price Below Lower Boundary	countBooksByPrice	To test out-of-bound low price	price = -1	book_count = 0	Validates negative prices are ignored.
Price Above Upper Boundary	countBooksByPrice	To test out-of-bound high price	price = 1,000,001	book_count = 0	Validates high out-of-bound prices.
Nominal Price	countBooksByPrice	To test typical case with valid data	price = 25	Count matches expected records.	Validates average-case scenario.

Class Evaluation Testing

fetchLibrarianSalesData()

This method was used for Equivalence class testing according to this idea. The method takes a librarian and two dates as inputs and gives the book names, prices, quantity in that given date range. A class is valid if the librarian exists and if the given data range has the correct format (startDate < endDate). Everything else should count as invalid.

Test Case	Reason	Input Value	Expected Output	Comments
Valid Date Range with Results	Ensures correct sales data is fetched	librarianUsername = "librarian1", startDate = 2024-01-01, endDate = 2024-01-31	List of sales records for January 2024.	Validates successful query execution.
Valid Date Range without Results	Ensures empty result set is handled gracefully	librarianUsername = "librarian1", startDate = 2024-01-01, endDate = 2024-01-31	Empty list.	Validates no results case.
Null Inputs	Ensures null checks trigger exceptions	librarianUsername = null, startDate = null, endDate = null	IllegalArgumentException: "Inputs cannot be null".	Validates input validation.
Incorrect Date Range	Ensures inverted date range triggers exception	startDate = 2024-02-01, endDate = 2024-01-01	IllegalArgumentException: "Start date cannot be after end date".	Validates input validation for date range.

Test Case	Reason	Input Value	Expected Output	Comments
Invalid Username	Ensures non-existent librarian returns no results	librarianUsername = "nonExistent"	Empty list.	Validates handling of invalid users.

Code Coverage Testing

fetchLibrarianSalesData()

The very same method used for EQ was used for code coverage too, due to its complexity. This was one of the few methods in our project which would be suitable for all types of code coverage methods we have learned so far. There are if-statements included, loops, try-catch-finally, database interactions (SQL commands). How the method works has been explained at EQ since it's the same.

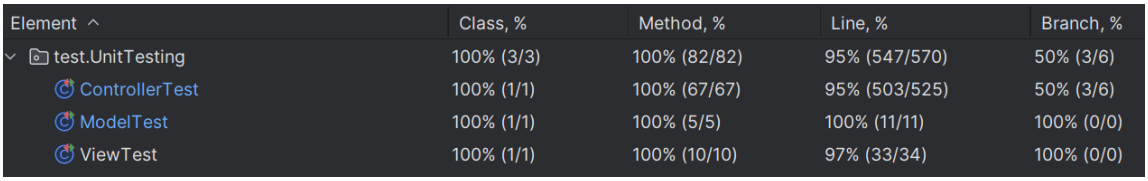
Test Case	Type of Coverage	Reason	Input Value	Expected Output	Comments
Statement Coverage	Statement Coverage	Ensures all statements execute	Valid inputs	List of sales records for given range.	Verifies full statement execution
Branch Coverage	Branch Coverage	Ensures all branches are covered (Valid, invalid, null)	Valid inputs, null inputs, invalid date range	Mixed outcomes: successful result, exceptions for invalid cases.	Validates all decision points.
Condition Coverage	Condition Coverage	Ensures all conditions are covered.	Valid inputs and various edge cases	Mixed outcomes: results for valid cases, exceptions for invalid ones.	Validates condition logic.
MCDC Coverage	MCDC Coverage	Ensures all combinations of conditions that could change the output are tested	Valid inputs and edge cases	Comprehensive outcomes: verifies condition combinations for correctness.	Covers all condition scenarios.

Unit Testing

Indrit was the member responsible for all Unit Testing.

The method I chose to use for my Unit Testing part was doing the table first, filling it up column by column until the result and only then I created the test cases. The testing method I chose to do was Black Box Testing because that's what fitted this projects methods the most. Most of the methods were tested with valid inputs, invalid inputs, edge cases and if the method has any SQL Exceptions then I also checked how will the method behave in case of those exceptios occurring. I used JUnit and Mockito to do these Unit Tests and mocked the database connection, the prepared statements and the result set utilizing those two libraries and created 76 unit tests with them. Out of all of them only 7 of them failed the tests, because the architecture of the management system was messy. Additionally, the original methods were also modified a little bit only when the change was minor. This includes if-statements in methods like: updateBookStock(), registerEmployees and validateCredentials().

This is also the Coverage that I was able to get with my tests.



Element	Class, %	Method, %	Line, %	Branch, %
test.UnitTesting	100% (3/3)	100% (82/82)	95% (547/570)	50% (3/6)
ControllerTest	100% (1/1)	100% (67/67)	95% (503/525)	50% (3/6)
ModelTest	100% (1/1)	100% (5/5)	100% (11/11)	100% (0/0)
ViewTest	100% (1/1)	100% (10/10)	97% (33/34)	100% (0/0)

Below are the methods that have been used for unit testing and their respective table values.

Method	Test Case Description	Input	Expected Output	Result	Test Case Reason
MODEL METHODS					
hashPassword()	Tested this method for valid input and added assertNotNuull and assertFalse if empty	'Password123'	A string with 64 letters and numbers	PASS	To test the method with valid inputs
hashPassword()	Tested this method for invalid characters in it	' !!!@@@...,,,***	Illegal Argument Exception	PASS	To test the method with invalid inputs
hashPassword()	Tested this method for empty input	"	IllegalArgument Exception	PASS	To test the method with empty inputs

VIEW METHODS					
validateISBN()	Tested this method with a valid format of ISBN	1234-56-7890	We expect no problems here, the program will work correctly	PASS	To test the method when it is given the correct inputs
validateISBN()	Tested this method with the invalid format on an ISBN	1234567890 123-45-6789 ABCD-56-7890	We expect this to tell us invalid format was used for the ISBN	PASS	To test the method when it is given the incorrect inputs
validateISBN()	Tested this method with empty values	“”	We expect this to tell us that empty ISBN is invalid	PASS	To test the method when it is given “” as input
validateAuthor()	Tested this method with a valid name and using assertTrue	Marie Curie	We expect no problems here, the program will work correctly	PASS	To test the method when it is given the correct inputs
validateAuthor()	Tested this method with an invalid name, and using assertFalse	!@#$%^	We expect this to tell us invalid format was used for the author name	PASS	To test the method when it is given the incorrect inputs
validateAuthor()	Tested this method with empty values and using assertFalse	“”	We expect this to tell us that empty author name is invalid	PASS	To test the method when it is given “” as input
processAddedBooks()	Adding books to the bill and then verifying if they are saved in the db	new Book("https://books.com", "1984", "FICTION", "2222-22-2222", "AUTHOR 1", 5) (createBillControllerMock).saveBillToDatabase(librarianUsername, "1984", 5, 25.0);	To pass with no problems	FAIL	To test the method with a number of books to see if their bills are added to the db

processAddedBooks()	No added bills for books, just an empty set	*verify*(createBillControllerMock, *never*()).saveBillToDatabase(*anyString*(), *anyString*(), *anyInt*(), *anyDouble*()	To not call the saveBillToDatabase()since it is empty	PASS	To test the method with an empty list
processAddedBooks()	Adding a bill for a book and then forcing saveBillToDatabase to throw an exception	new RuntimeException("Database error")	To attempt the saveBillToDB() but not actually run it	FAIL	To test the method with a book and then throwing an error to see how it handles exceptions
CONTROLLER METHODS					
bookExistsByName()	Tested with an existing book	1984	Returns true	PASS	To make sure the method works correctly
bookExistsByName()	Tested with a non existing book	1985	Returns false	PASS	To make sure the method works correctly
bookExistsByName()	Tested with the title field empty	“	Returns no book data	PASS	To make sure the method works correctly
getBookByName()	Tested with a book title existing in the db	1984	Returns the book data	PASS	To check how the method behaves with correct inputs
getBookByName()	Tested with a book title not existing in the db	1994	Returns no data	PASS	To check how the method behaves with incorrect inputs
getBookByName()	Tested with an empty title	“”	Returns no data	PASS	To check how the method

					behaves with edge case
updateBookStock()	Tested with valid ISBN and stock number, used when for db mocking and verify to ensure they're updated	9781-23-4567 , 50	Passes the tests and updates the book stock	PASS	To test the method with valid inputs
updateBookStock()	Tested with -1, with an invalid input	9781234567, -1	Should say that negative input isn't valid	PASS	To see how the method responds to invalid inputs
updateBookStock()	Tested with value equal to quantity itself (max value)	9781-23-4567, 100 (When the book stock is 100)	It should pass the test with no problems, however if UI was also included we'd get an alert saying stock is low	PASS	To see how the methods logic responds to large number of stock
saveBillToDatabase()	Tested with valid inputs uitilizing verify method from mockito	Librarian1, Harry Potter, 20, 19.99	Successful insertion	PASS	To test the methods interacti on with the db
saveBillToDatabase()	Tested with invalid inputs such as negative quantity	Librarian1, Harry Potter, -5, 19.99	Unsuccessful insertion due to invalid inputs	PASS	To see how the method responds to invalid inputs
getAllISBNOrderedByStock ()	Tested with existing ISBNS	Takes no inputs, but in the db there are 10 isbn	Successfully obtaining all ISBNs	PASS	To test the methods with valid ISBNs
getAllISBNOrderedByStock ()	Tested with no ISBN in db	Takes no input, but in the db there are 0 isbn	Obtains 0 of them	PASS	To test the method with 0 ISBNs
getAllISBNOrderedByStock ()	Tested with db connectivit y errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problem

					s with db
fetchLibrariansFromDatabase()	Tested with existing librarians	Takes no inputs, but in the db there are existing librarians	Successfully obtaining all ISBNs	PASS	To test the methods with valid librarians
fetchLibrariansFromDatabase()	Tested with no librarians in db	Takes no input, but in the db there are 0 librarians	Obtains 0 of them	PASS	To test the method with 0 librarians
fetchLibrariansFromDatabase()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
getDailyStatistics()	Tested with a number of daily stats	Takes no inputs, but in the db it has multiple daily stats	Successfully obtains the daily statistics	PASS	To test the method with valid stats
getDailyStatistics()	Tested with 0 daily stats	Takes no inputs, but in the db there is 0 daily stats	Obtains 0 of them	PASS	To test the method with 0 stats
getDailyStatistics()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
getMonthlyStatistics()	Tested with a number of monthly stats	Takes no inputs, but in the db it has multiple monthly stats	Successfully obtains the monthly statistics	PASS	To test the method with valid stats
getMonthlyStatistics()	Tested with 0 monthly stats	Takes no inputs, but in the db there is 0 monthly stats	Obtains 0 of them	PASS	To test the method with 0 stats
getMonthlyStatistics()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when

					there are problems with db
getTotalStatistics()	Tested with the total stats	Takes no inputs, but in the db it has the total stats	Successfully obtains the statistics	PASS	To test the method with valid stats
getTotalStatistics()	Tested with 0 stats	Takes no inputs, but in the db there is 0 stats	Obtains 0 of them	PASS	To test the method with 0 stats
getTotalStatistics()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
getStatistics()	Tested with the valid query	"SELECT book_title, SUM(quantity) as total_quantity, SUM(total_price) as total_price "+"FROM bills GROUP BY book_title"	Successfully obtains the statistics	PASS	To test the method with valid query
getStatistics()	Tested with 0 stats	"book_title, _price"+" bills GROUP BY book_title"	Obtains 0 statistics	PASS	To test the method with 0 stats
getStatistics()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
registerEmployee()	Test with valid inputs	Librarian, password, librarian, 23/02/2000, 0683894567,librarian@gmail.com, 50000, librarian	Successfully registers the employee	FAILS	To test the methods behavior with valid inputs
registerEmployee()	Test with duplicate name	Librarian, password, librarian, 22/07/1997, 0683893407,libra	Doesn't register the employee.	FAILS	To test how the method behaves with

		rian90@gmail.com, 50000, librarian			duplicate names
registerEmployee() (EQ Testing) Modified method	Test with valid age values	Librarian, password, librarian, 23/02/2000, 0683894567,librarian@gmail.com, 50000, librarian	Successfully registers the employee	FAILS	To test with valid age group
registerEmployee()	Test with underage values	Librarian, password, librarian, 23/02/2017, 0683894567,librarian@gmail.com, 50000, librarian	Doesn't register the employee.	FAILS	To test with underage values
registerEmployee()	Test with overage values	Librarian, password, librarian, 23/02/1955, 0683894567,librarian@gmail.com, 50000, librarian	Doesn't register the employee.	FAILS	To test with overage values
registerEmployee()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
deleteUser()	Tested with valid option	Takes no inputs, but in the db it has the total stats	Successfully obtains the statistics	PASS	TO test the methods behavior with valid option
deleteUser()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
getUsersByRole()	Tested with valid roles	librarian	Librarian	PASS	To test the methods behavior with valid input
getUsersByRole()	Tested with 0 roles in db	librarian	Nothing gets obtain form the db	PASS	To test the methods

					behavior with valid input
getUsersByRole()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
updateUser()	Tested with valid inputs	Librarian, password, librarian, 23/02/2000, 0683894567,librarian@gmail.com, 50000, librarian, Librarian2	Successfully updated user	PASS	To test the method with valid inputs
updateUser()	Tested with non-existing user	Librarian, password, librarian, 23/02/2000, 0683894567,librarian@gmail.com, 50000, librarian, Librarian100	No update occurs	PASS	To test the method with non-existing user to see its behavior
updateUser()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
isUsernameExists()	Tested with valid inputs	Librarian1	True	PASS	To test the method with valid inputs
isUsernameExists()	Tested with invalid inputs	Librarian50	False	PASS	To test the method with non-existing username to see its behavior
isUsernameExists()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior

					when there are problems with db
getTotalBillCostsAddedOnDate()	Tested with bills that day	01/18/2025	The amount of costs for that day	PASS	To test the method with bills made that day
getTotalBillCostsAddedOnDate()	Tested with no bills that day	01/01/2025	The amount of cost for that day	PASS	To test the method with 0 bills that day
getTotalBillCostsAddedOnDate()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
getTotalBillCostsAddedInRange()	Tested with valid date range	Yearly	Passes and shows the cost	PASS	To test the methods behavior with correct input
getTotalBillCostsAddedInRange()	Tested with invalid date range	01/18/2025 – 01.18.2024	Doesn't not show the cost	PASS	To test the behavior of the method with invalid inputs
getTotalBillCostsAddedInRange()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db
validateCredentials()	Tested with correct username	Librarian1, librarian1	Passes the tests	PASS	Testing the method with

	and password				valid credentials
validateCredentials()	Tested with incorrect username	Librariaaaan1, librarian1	Fails the tests	PASS	Testing the method with invalid credentials(username)
validateCredentials()	Tested with incorrect password	Librarian1, libraaaarian1	Fails the tests	PASS	Testing the method with invalid credentials(password)
validateCredentials()	Tested with db connectivity errors	Takes no inputs but there is a problem with db connection	SQLException	PASS	To test the methods behavior when there are problems with db

Integration Testing

Borian was the member responsible for all Integration Testing.

For the integration tests in the software, a **bottom-up approach** was adopted. This methodology involved starting the testing process at the lower-level modules, such as data access and backend services, and incrementally integrating and testing higher-level components, culminating in the full integration of the application.

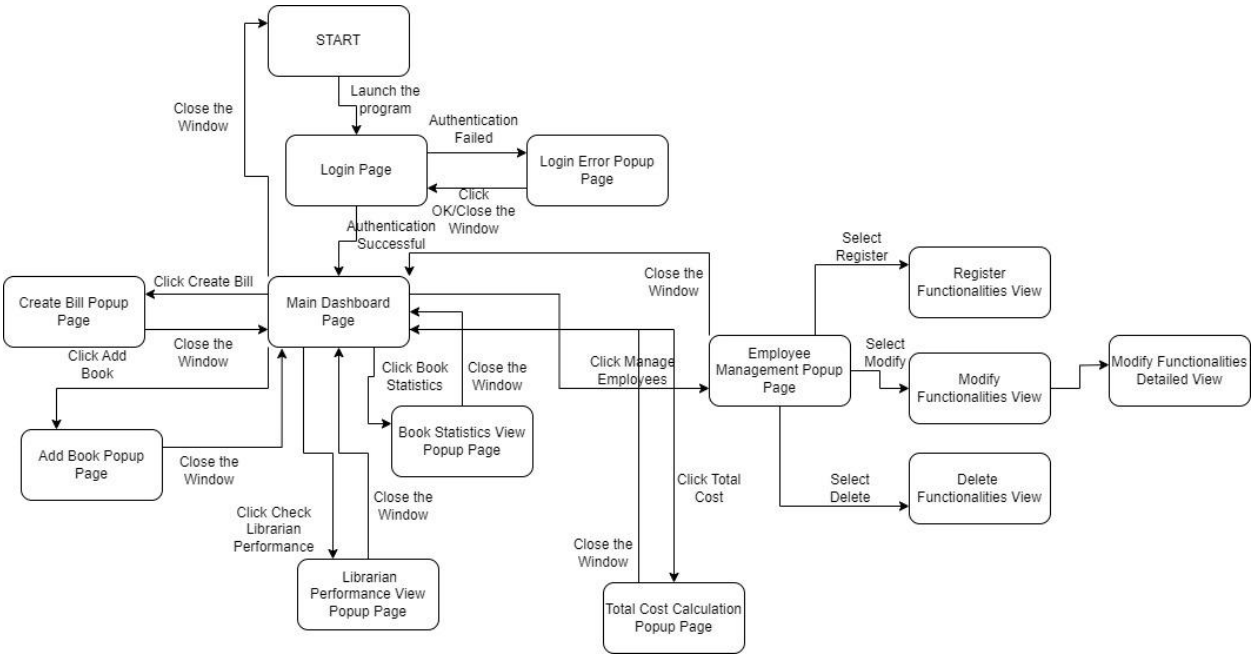
The Model-View-Controller (MVC) architecture is inherently modular, making the bottom-up approach particularly suitable for the following reasons. Focus on Core Logic First (Model Layer)The Model layer, which contains the application's data and business logic, is foundational in MVC. By starting with integration tests at this layer, we ensure the core logic is robust and reliable before moving to higher layers like the Controller or View. Incremental Confidence Building ,by thoroughly testing backend services and gradually integrating Controllers and Views, we can identify and resolve issues early. This reduces the likelihood of debugging complex interactions across the entire stack later. Decoupled Layer Testing, MVC architecture emphasizes separation of concerns. Testing lower layers independently ensures each module is functioning correctly before it interacts with other components, minimizing integration errors. Testing lower layers first helps detect issues at the foundational level, avoiding redundant testing of higher layers until they are fully dependent on stable components.

Integration Test Class	**Integration Test Cases**	**Explanation**
AddBookIT	1)testValidBookSubmission 2)testinvalidISBNAlert 3)testIvalidAuthorAlert 4)testIvalidSellingPriceAlert	Ensures the Add Book functionality validates input correctly and handles data interactions reliably.
BillIT	1)testLoadBookInfo 2)testAddValidBook 3)testAddInvalidQuantity	Tests the billing functionality, ensuring smooth integration of data retrieval and user input validation.
BookStatisticsIT	1)testBookStatistcsView_ showscorrectDataWhen SelectingTimeline	Validates that the statistics view displays accurate and up-to-date data when filters or timelines are applied.
CheckLibrarianPerformanceIT	1)TestLibrarianPerformanceView	Ensures the performance view of librarians is displayed correctly, with accurate data aggregation.

LoginControllerIntegrationTest	1)testLoginSuccessfull 2)testLoginFailure	Verifies successful and failed login scenarios, ensuring user authentication interacts correctly with the backend.
ManageEmployeesIntegrationTest	1)testEmployeeRegistration 2)TestEmployeeDeletion 3)TestEmployeeModification 4)testValidation	Tests employee management operations, including registration, deletion, modification, and input validation.
TotalCostIntegrationTest	1)testCalculateTotalCost	Validates the calculation of total costs, ensuring accurate data aggregation and processing.
CreateBillControllerIT	1)testgetBookByISBN_ValidIsbn 2)testGetBookByISBN_InvalidIsbn 3)testUpdateBookStock 4)testSaveBillToDatabase 5)testAllIsbnsOrderedBySTock	Ensures the bill creation process integrates correctly, from book retrieval to stock updates and bill saving.

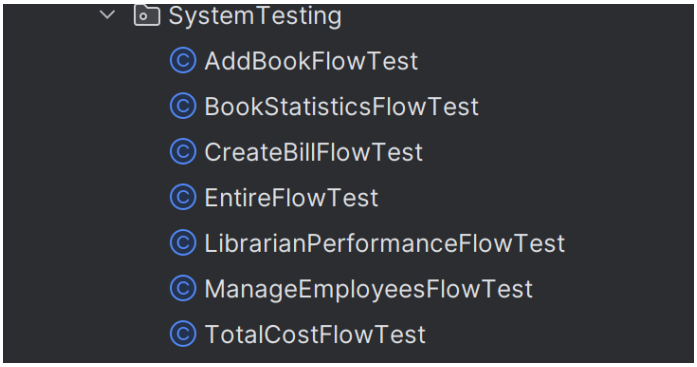
System Testing

Engjëll was the member responsible for all System Testing.



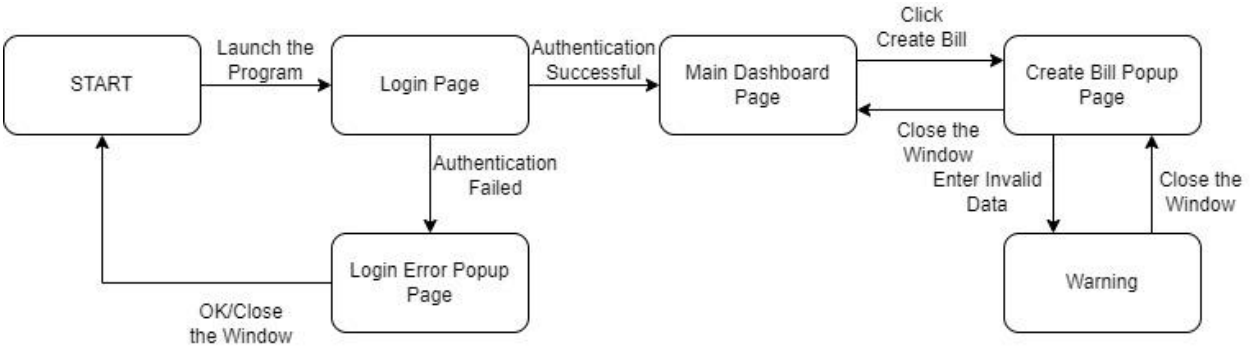
The diagram above represents the Final State Machine which shows the most important states of the program, with the respective events that cause state transitions.

Note: The states “Register Functionalities View”, “Modify Functionalities View”, “Delete Functionalities View” and “Modify Functionalities Detailed View” return to “Employee Management Popup Page” when their windows are closed. These transitions are not shown here for visualization purposes.



The image to the left shows all the classes that were built for system testing.

Flow 1 – CreateBillFlowTest.java



Test Case 1

The **testCreateBill** method is designed to validate the entire flow of successfully creating a bill within the application. It begins by simulating a user logging in with valid credentials and then navigating to the "Create Bill" section of the application. Once there, it selects an ISBN from a combo box, loads the corresponding book information, enters a valid quantity for the book, and adds the book to the bill using the "Add Book" button. Finally, it simulates clicking the "Create Bill" button to complete the bill creation process and asserts that the bill table is cleared afterward. This test verifies the "happy path," ensuring that the system behaves as expected when all inputs are valid and the user follows the correct workflow.

Test Case 2

The **testInvalidQuantityAlert** method tests the system's response to invalid inputs, specifically a non-numeric quantity. The test begins by logging in with valid credentials and navigating to the "Create Bill" section. After selecting an existing ISBN and loading its information, it simulates entering an invalid, non-numeric quantity (e.g., letters) in the quantity field. When the user attempts to add the book with this invalid quantity, the test verifies that the system displays an alert popup, warning the user about the invalid input. This ensures that the application enforces proper data validation for quantity input.

Test Case 3

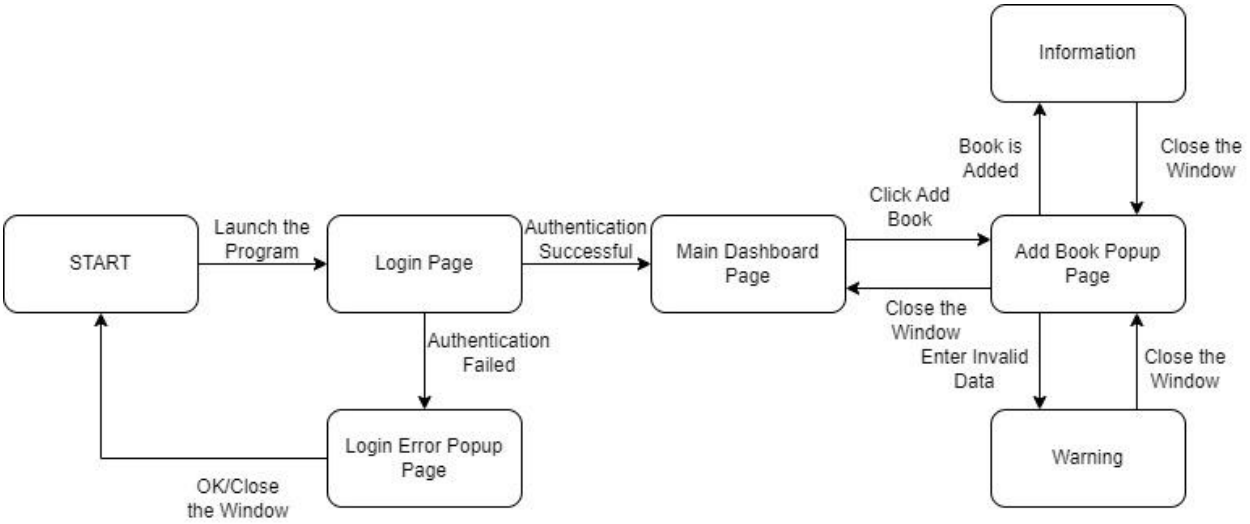
The **testInsufficientStockAlert** method checks the system's behavior when a user attempts to order a quantity that exceeds the available stock for a book. The test logs in with valid credentials, navigates to the "Create Bill" section, selects an existing ISBN, and loads its information. It then simulates entering a quantity that exceeds the stock limit and clicking the "Add Book" button. The test confirms that the system displays an appropriate alert popup, notifying the user about the insufficient stock. This ensures that the application prevents users from creating bills with unrealistic quantities.

Test Case 4

Finally, the **testLoginErrorWithWrongCredentials** method verifies the system's handling of incorrect login attempts. It simulates entering invalid credentials (e.g., a wrong username and

password) and clicking the login button. The test asserts that an alert popup appears, informing the user of invalid credentials. This ensures that the system properly restricts access to unauthorized users and provides clear feedback on login errors.

Flow 2 – AddBookTest.java



Test Case 1

The **testAddBook** verifies the ability to add a book successfully. The test starts by simulating user login with valid credentials, navigates to the "Add Book" section, and asserts the view is displayed. It fills book details such as URL, name, category, ISBN, author, and price before clicking the "Add Book" button. The test asserts that an alert popup appears, indicating the book is added.

Test Case 2

The **testInvalidSellingPrice** focuses on input validation for selling price. It simulates login and navigation to the "Add Book" section. After entering book details with a negative price, it asserts the alert popup's presence and validates its content, ensuring error messages display correctly for invalid input.

Test Case 3

The **testInvalidISBNFormat** checks the application's handling of an invalid ISBN. Following successful login and navigation to the "Add Book" section, it inputs an improperly formatted ISBN and asserts the error popup is displayed with the appropriate message.

Test Case 4

The **testInvalidAuthorName** validates the author field. After login and navigating to the "Add Book" section, it leaves the author field empty while providing other details. The test asserts that the alert popup appears with an appropriate error message.

Test Case 5

The **testBookSavedSuccessfully** simulates adding a valid book. Following login and navigation, valid details are entered, and the "Add Book" button is clicked. The test verifies an alert popup is displayed, indicating the book was added successfully.

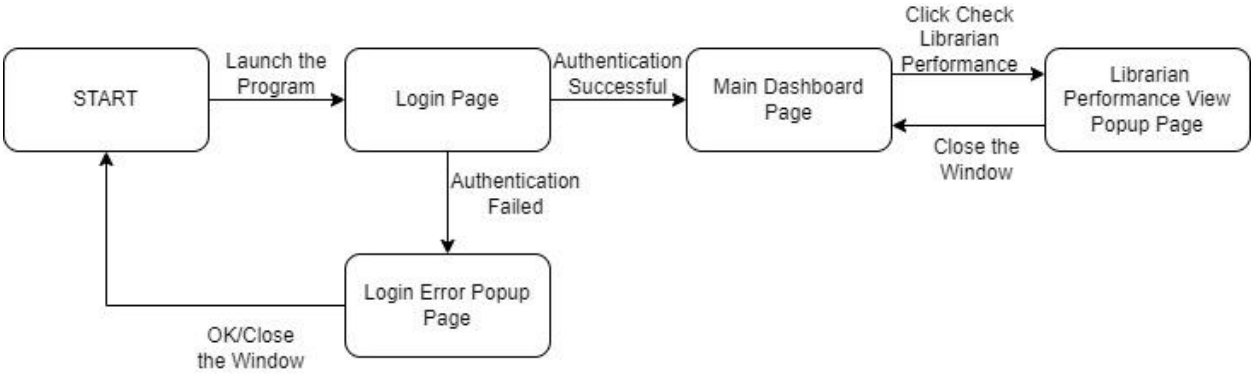
Test Case 6

The **testBookNotFound** handles cases where a nonexistent book's ISBN is provided. The test ensures that after inputting a nonexistent ISBN, the system displays an appropriate alert message.

Test Case 7

Finally, the **testEmptyISBNField** validates the system's behavior when the ISBN field is left empty. Similar to other tests, it simulates login, navigation, and data entry, leaving the ISBN blank. The test confirms the presence of an alert popup with a descriptive error message.

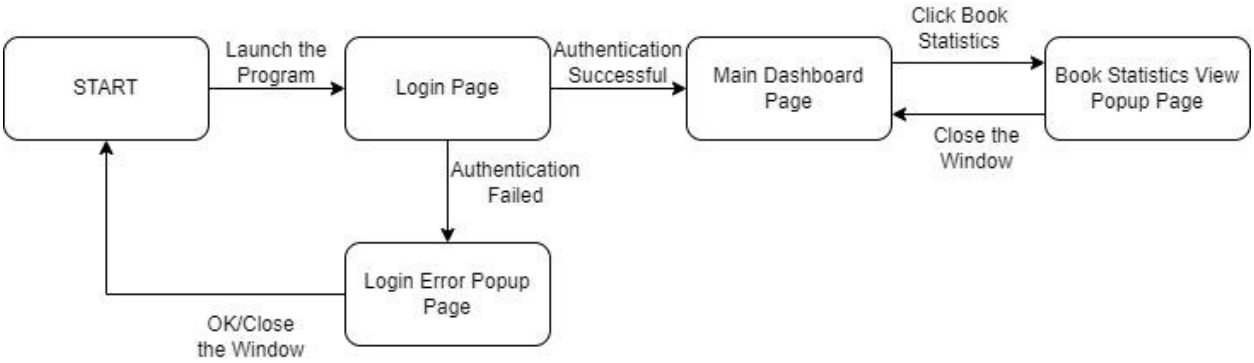
Flow 3 – LibrarianPerformanceTest.java



Test Case 1

The **testLibrarianPerformance** validates the process of checking a librarian's performance over a specific period. The test begins by simulating user login with valid credentials and verifies the login is successful. After logging in, it navigates to the "Check Librarian Performance" section via the UI and asserts that the correct view is displayed. It selects a librarian from a dropdown menu, inputs start and end dates for the performance evaluation period, and clicks the "Show Data" button to display the results. The test concludes by asserting that the performance data for the selected librarian is displayed, confirming the functionality works as intended.

Flow 4 – BookStatisticsFlowTest.java



Test Case 1

The **testBookStatistics** verifies the functionality to display overall book statistics. The test starts by simulating a user login with valid credentials and asserts that the login is successful. After logging in, the test navigates to the "Book Statistics" section and verifies that the corresponding view is displayed. It selects the "Total" option from the statistics dropdown menu and checks that the book statistics table is present and contains rows, ensuring the functionality to display total book statistics works correctly.

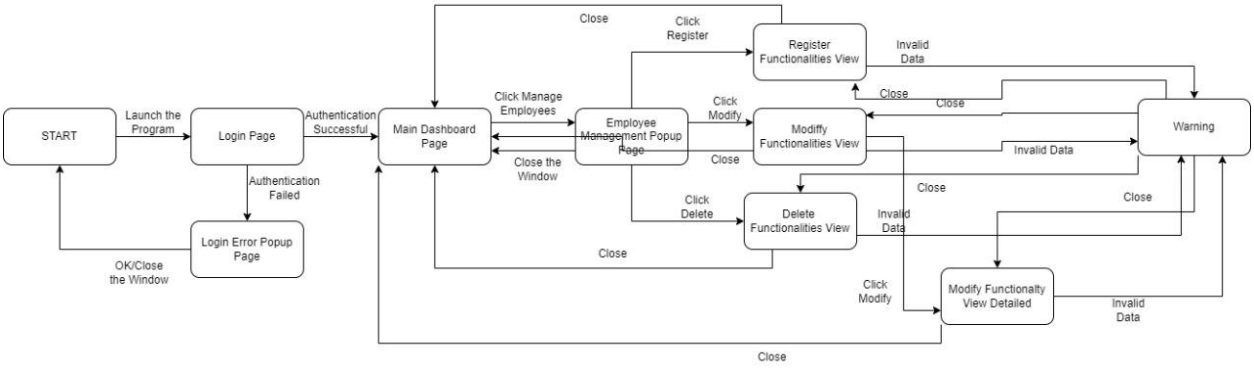
Test Case 2

The **testBookStatisticsDaily** verifies the functionality to display daily book statistics. The test begins by simulating a user login with valid credentials and confirming the login's success. The test then navigates to the "Book Statistics" section, ensuring the view is displayed. It selects the "Daily" option from the statistics dropdown menu and asserts that the book statistics table appears with rows, confirming the system correctly displays daily book statistics.

Test Case 3

The **testBookStatisticsMonthly** ensures that monthly book statistics are displayed accurately. The test starts by simulating a user login with valid credentials, verifying the login process is successful. After navigating to the "Book Statistics" section, it validates the presence of the view. The test then selects the "Monthly" option from the statistics dropdown and verifies the book statistics table is displayed with rows, demonstrating the system's ability to correctly show monthly book statistics.

Flow 5 – ManageEmployeesFlowTest.java



Test Case 1

The **testRegisterCorrect** method tests the registration functionality in the "Manage Employees" section. It simulates a successful login with valid credentials ("admin1"), navigates to the "Manage Employees" section, and attempts to register a new employee. The test includes interactions with the form, where valid data is entered, followed by clicking the register button. Afterward, it verifies that an alert popup is shown to confirm the success of the operation. This test verifies that the registration functionality works correctly with proper inputs.

Test Case 2

The **testModifyCorrect** method verifies the "Modify Employee" functionality. After logging in with valid credentials, it navigates to the "Manage Employees" section and selects the "Modify" option. The test simulates modifying an employee's details, such as selecting a manager and librarian, changing the salary, and clicking the "Update" button. It then checks whether an alert popup is displayed, indicating the success of the modification. This test ensures that the modify functionality is working as expected with valid input data.

Test Case 3

The **testDeleteCorrect** method tests the functionality of deleting an employee. After logging in and navigating to the "Manage Employees" section, it selects the "Delete" option and chooses an employee to delete. Once the "Delete" button is clicked, it checks whether an alert popup appears, confirming the deletion. This test ensures that the delete functionality behaves as expected when the user selects an employee to remove.

Test Case 4

The **testRegisterInvalidEmail** method tests the registration functionality when an invalid email is provided. After logging in, the test navigates to the "Manage Employees" section and attempts to register a new employee with an invalid email address ("invalid-email"). Upon clicking the

register button, the test checks for the presence of an alert popup, indicating that the system correctly identifies and handles invalid email addresses during registration.

Test Case 5

The **testRegisterInvalidPassword** method validates the registration process with an invalid password. The test simulates logging in, then navigates to the registration form, entering an empty password. After clicking the register button, the test verifies that the system shows an alert popup, confirming that the system correctly handles empty or invalid password fields during registration.

Test Case 6

The **testRegisterInvalidUsername** method tests the registration functionality when an invalid (empty) username is entered. It follows the same process as the previous tests but focuses on entering an empty string for the username field. The test ensures that the system displays an alert popup when an invalid username is provided, verifying that the system validates the username field properly.

Test Case 7

The **testRegisterInvalidName** method ensures that the system correctly handles the case when the name field is left empty during employee registration. The test simulates filling out the form with valid information for all fields except the name, which is left empty. After attempting to register, it checks for an alert popup, confirming that the system detects the empty name field and displays an appropriate error message.

Test Case 8

The **testRegisterInvalidPhone** method tests the registration functionality with an invalid phone number. The test involves filling out the registration form with a non-numeric or improperly formatted phone number ("invalid-phone"). After submitting the form, it checks for an alert popup, ensuring that the system properly handles invalid phone numbers during registration by displaying an error message.

Test Case 9

The **testRegisterInvalidSalary** method checks the registration functionality when an invalid salary value is entered. The test simulates entering a non-numeric or otherwise invalid salary ("invalid-salary") during the employee registration process. After attempting to register, the test verifies that the system shows an alert popup, indicating that the salary field has not been correctly filled out, ensuring proper validation.

Test Case 10

The **testModifyInvalidSalary** method verifies the "Modify Employee" functionality when an invalid salary is entered. After selecting an employee and editing their salary to a non-numeric value ("invalid"), the test checks if the system displays an alert popup, confirming that the system detects invalid salary inputs and provides appropriate feedback to the user.

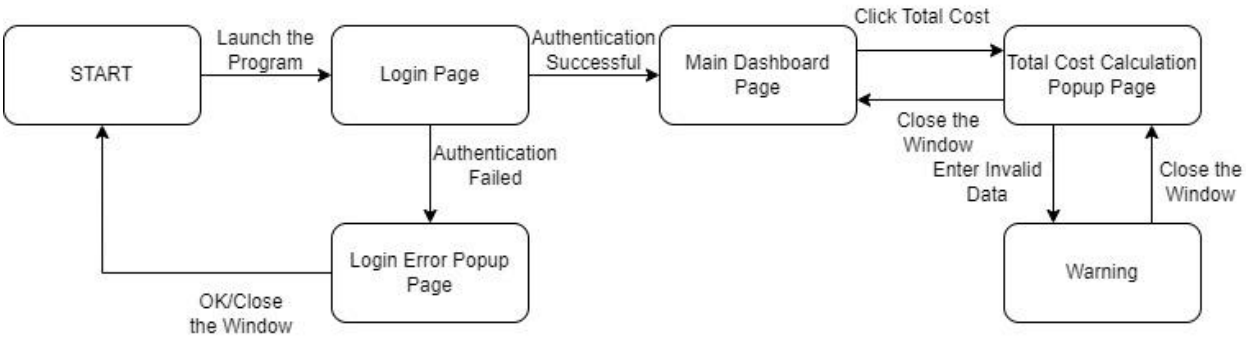
Test Case 11

The **testModifyEmptyFields** method tests the behavior of the "Modify Employee" feature when required fields, such as salary, are left empty. The test simulates modifying an employee's data but intentionally erases the salary field. After submitting the form, the test verifies that the system detects the missing information and displays an alert popup, ensuring that empty fields are properly validated and flagged.

Test Case 12

The **testDeleteUserNotSelected** method tests the delete functionality when no user is selected for deletion. After logging in and navigating to the "Manage Employees" section, the test simulates attempting to delete a user without selecting one first. Upon clicking the "Delete" button, the test checks for an alert popup, ensuring that the system alerts the user that no employee has been selected for deletion, preventing the action from proceeding without a valid selection.

Flow 6 – TotalCostFlowTest.java



Test Case 1

testTotalCost simulates the process of logging in with the username admin1 and password admin1. After logging in, the test navigates to the "Total Cost" section from the combo box, selects the "Daily" option from the timeframe dropdown, and then clicks the "Calculate" button to compute the total cost. The test then verifies that the book statistics table is displayed and contains rows, ensuring that the calculation has been successfully performed for the daily timeframe.

Test Case 2

Similar to the testTotalCost method, **testTotalCostWeekly** simulates logging in with the same credentials (admin1/admin1). It then navigates to the "Total Cost" section and selects the "Weekly" option from the timeframe combo box. After clicking the "Calculate" button, it checks if the book statistics table is present and contains rows, ensuring that the total cost calculation was successfully performed for the weekly timeframe.

Test Case 3

In **testTotalCostMonthly**, the user logs in with the credentials admin1/admin1 and selects the "Total Cost" section from the combo box. After selecting the "Monthly" option from the timeframe dropdown and clicking the "Calculate" button, the test verifies that the book statistics table is present and has rows. This ensures that the calculation of total cost for the monthly timeframe is functioning correctly.

Test Case 4

testTotalCostYearly follows a similar structure to the previous tests. The test simulates logging in with the credentials admin1/admin1, navigates to the "Total Cost" section, and selects the "Yearly" option from the timeframe combo box. Upon clicking the "Calculate" button, it ensures that the book statistics table is displayed and contains rows, validating that the yearly total cost calculation works as expected.

Flow 7 – EntireFlowTest.java

This is one big test which tests a complete execution of the program, from logging in to visiting all the other windows tested on the previous flows.

All of these tests passed successfully.