PROJECT REPORT

Software Testing and Quality Assurance

22/12/2023

Worked By:

Kevin Llaca

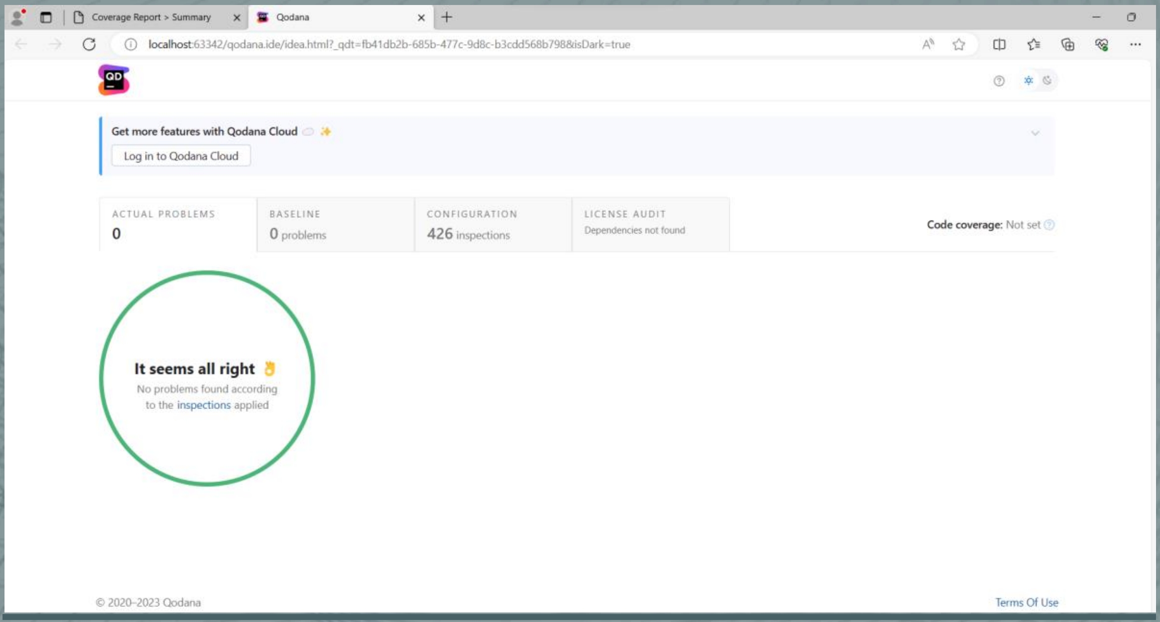
Tea Malasi

GitHub link: https://github.com/tmalasi/tamam.git

Static Testing

Quodana

For static testing, we employed Quodana, a static analysis tool. Quodana helped us analyze the source code of our application without executing it. It identified potential issues, coding standards violations, and other static code-related problems, allowing us to proactively address them during the development process. This static analysis contributed to improving the overall code quality and ensuring that our code adheres to best practices. Quodana report screenshot:



Coverage Report > Summary
Qodana
localhost:63342/qodana.ide/idea.html?_qdt=fb41db2b-685b-477c-9d8c-b3cdd568b798&isDark=true
QD
Log in to Qodana Cloud
Terms Of Use

Get more features with Qodana Cloud

ACTUAL PROBLEMS	BASELINE	CONFIGURATION	LICENSE AUDIT
0	0 problems	426 inspections	Dependencies not found

Code coverage: Not set

It seems all right

No problems found according to the inspections applied

© 2020–2023 Qodana

Testing the code

The testing phase is a critical component of the software development lifecycle, ensuring that the implemented code meets specified requirements and functions as intended. In this report, we present the results of our rigorous testing efforts focused on two distinct components of our project: static testing using Quodana and dynamic testing through JUnit. Our dynamic testing is centered around the JUnit testing framework, where we have meticulously crafted test cases to assess the functionality, correctness, and reliability of our software components. In this report is:

- Specification Based testing: Boundary Value Testing and Equivalence Testing
- Code based Testing: Ensuring we have 100%-line coverage
- Mock Testing
- Integration Testing
- Basis path Testing
- JavaFx Testing
- All test coverage percentage
- Mutation Testing
- Work done by everyone

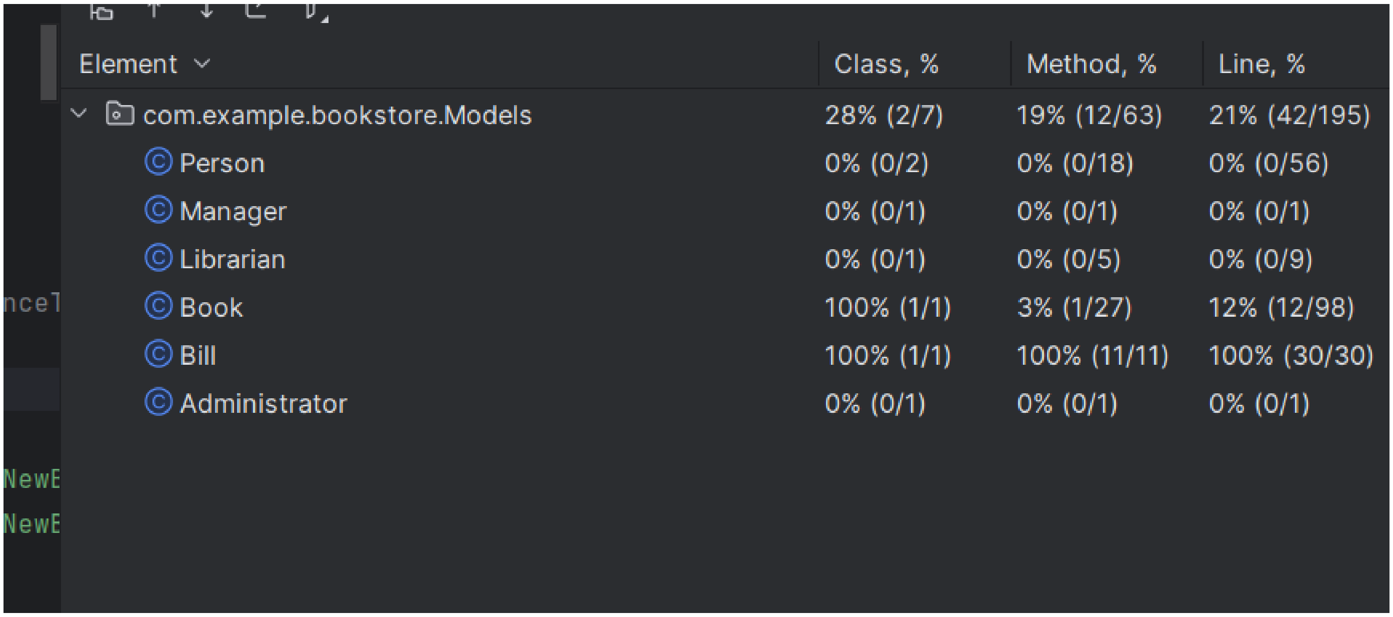
Testing Bill Model class

HOW have we divided the equivalence testing and boundary testing?

- Equivalence testing is applied by considering different classes of input for each attribute (e.g., null, valid, invalid).
- Boundary testing is applied by testing values at the edges of the valid input range for quantity and total amount for example. Everything that has a range in the setters its tested by boundary testing.

The combination of equivalence testing and boundary testing helps ensure comprehensive coverage of input scenarios and increases the likelihood of discovering potential issues. The test cases are well-organized and cover a diverse set of scenarios. This comprehensive set of test cases enhances the reliability and robustness of the class. We have done this for all the Models.

Coverage of the Bill Model.



Element	Class, %	Method, %	Line, %
com.example.bookstore.Models	28% (2/7)	19% (12/63)	21% (42/195)
Ⓒ Person	0% (0/2)	0% (0/18)	0% (0/56)
Ⓒ Manager	0% (0/1)	0% (0/1)	0% (0/1)
Ⓒ Librarian	0% (0/1)	0% (0/5)	0% (0/9)
Ⓒ Book	100% (1/1)	3% (1/27)	12% (12/98)
Ⓒ Bill	100% (1/1)	100% (11/11)	100% (30/30)
Ⓒ Administrator	0% (0/1)	0% (0/1)	0% (0/1)

Testing the methods:

Setter Method: void setBooks(ArrayList<Book> books)
Description: Sets the internal 'books' field of the Bill class to the provided non-null ArrayList of Book objects; throws an IllegalArgumentException if the input parameter is null, ensuring data integrity.

Test Name	Description of Test	Test Type
testSetBooksValidValue	Tests the setBooks method with a valid list of books.	Boundary Value Testing and Equivalence Testing
testSetBooksNull	Tests the setBooks method with a null list of books.	Boundary Value Testing and Equivalence Testing

Setter Method: void setBillId(String billId)
Description: Sets the unique bill ID, validating that it is not null and does not exceed a specified maximum length; throws IllegalArgumentException if the conditions are not met.

Test Name	Description of Test	Test Type
testSetBillIdValidValue	Tests the setBillId method with a valid bill ID.	Boundary Value Testing and Equivalence Testing
testSetBillIdNullValue	Tests the setBillId method with a null bill ID.	Boundary Value Testing and Equivalence Testing
testSetBillIdMinAndMaxValue	Parameterized test for setting bill ID with both minimum and maximum valid values.	Boundary Value Testing and Equivalence Testing
testSetBillIdLessThanMinAndMoreThanMax	Parameterized test for setting bill ID with values less than the minimum and more than the maximum.	Boundary Value Testing and Equivalence Testing

Setter Method: void setQuantity(int quantity)
Description: Sets the quantity of books, ensuring it is within a specified valid range (between 1 and 10); throws IllegalArgumentException if the quantity is outside this range.

Test Name	Description of Test	Test Type
testSetQuantityValidValue	Tests the setQuantity method with a valid quantity.	Boundary Value Testing and Equivalence Testing
testSetQuantityMinAndMaxValue	Parameterized test for setting quantity with both minimum and maximum valid values.	Boundary Value Testing and Equivalence Testing
testSetQuantityLessThanMinAndMoreThanMax	Parameterized test for setting quantity with values less than the minimum and more than the maximum.	Boundary Value Testing and Equivalence Testing

Setter Method: void setTotalAmount(double totalAmount)
Description: Sets the total amount, ensuring it is non-negative; throws IllegalArgumentException if the total amount is negative.

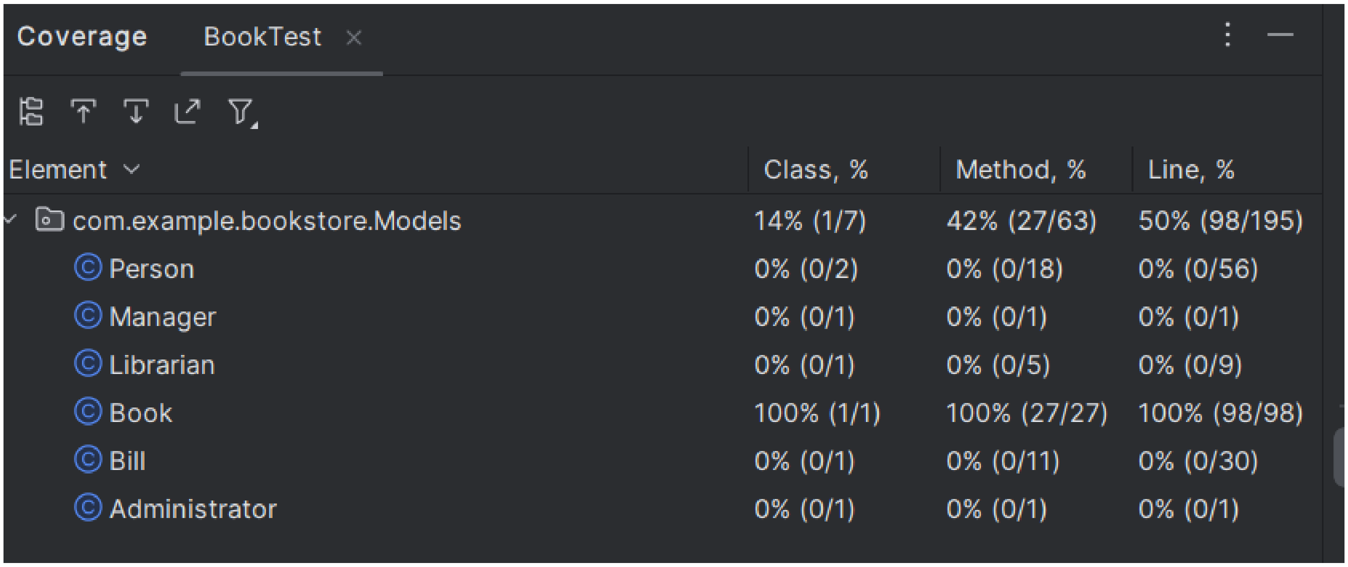
Test Name	Description of Test	Test Type
testSetTotalAmountValidValue	Tests the setTotalAmount method with a valid total amount.	Boundary Value Testing and Equivalence Testing
testSetTotalAmountMinValue	Tests the setTotalAmount method with the minimum valid total amount.	Boundary Value Testing and Equivalence Testing
testSetTotalAmountLessThanMin	Tests the setTotalAmount method with a total amount less than the minimum.	Boundary Value Testing and Equivalence Testing

Setter Method: void setDateOfTransaction(String dateOfTransaction)
Description: Sets the date of the transaction, ensuring it is not null; throws IllegalArgumentException if the date is null.

Test Name	Description of Test	Test Type
testSetDateOfTransactionValidValue	Tests the setDateOfTransaction method with a valid date of transaction.	Boundary Value Testing and Equivalence Testing
testSetDateOfTransactionNull	Tests the setDateOfTransaction method with a null date of transaction.	Boundary Value Testing and Equivalence Testing

Testing Book Model class

Coverage of the Book Model.



Element	Class, %	Method, %	Line, %
com.example.bookstore.Models	14% (1/7)	42% (27/63)	50% (98/195)
Person	0% (0/2)	0% (0/18)	0% (0/56)
Manager	0% (0/1)	0% (0/1)	0% (0/1)
Librarian	0% (0/1)	0% (0/5)	0% (0/9)
Book	100% (1/1)	100% (27/27)	100% (98/98)
Bill	0% (0/1)	0% (0/11)	0% (0/30)
Administrator	0% (0/1)	0% (0/1)	0% (0/1)

Testing the methods:

Setter Method: void setCategory(String category)
Description: Sets the category of the book, ensuring it is not null and does not exceed a specified maximum length.

Test Name	Description of Test	Test Type
testSetCategoryWhenItIsNull	Verifies that setting the category to null throws an IllegalArgumentException.	Equivalence Testing
testSetCategoryWhenItIsSmallerThanMin	Verifies that setting a category smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetCategoryWhenItsMinimumValidLength	Verifies setting a category with the minimum valid length.	Boundary Value Testing, Equivalence Testing

testSetCategoryWhenNormalLength	Verifies setting a category with a normal length.	Boundary Value Testing, Equivalence Testing
testSetCategoryWhenCloseTOMaxLength	Verifies setting a category close to the maximum length.	Boundary Value Testing, Equivalence Testing
testSetCategoryWhenMoreThanMaxLength	Verifies that setting a category longer than the maximum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setIsbn(String isbn)
Description: Sets the ISBN of the book, ensuring it is not null and does not exceed a specified maximum length.

Test Name	Description of Test	Test Type
testSetISBN_WithNullISBN	Verifies that setting ISBN to null throws an IllegalArgumentException.	Equivalence Testing
testSetISBNWhenItIsSmallerThanMin	Verifies that setting an ISBN smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetISBNWhenItsMinimumValidLength	Verifies setting an ISBN with the minimum valid length.	Boundary Value Testing, Equivalence Testing
testSetISBN_WhenNormalLength	Verifies setting an ISBN with a normal length.	Boundary Value Testing, Equivalence Testing
testSetISBN_WhenCloseToMaxLength	Verifies setting an ISBN close to the maximum length.	Boundary Value Testing, Equivalence Testing
testSetISBN_WhenMoreThanMaxLength	Verifies that setting an ISBN longer than the maximum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setTitle(String title)
Description: Sets the title of the book, ensuring it is not null and does not exceed a specified maximum length.

Test Name	Description of Test	Test Type
testSetTitle_WithNullTitle	Verifies that setting the title to null throws an IllegalArgumentException.	Equivalence Testing
testSetTitle_WhenLessThanMinLength	Verifies that setting a title smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetTitle_WhenCloseToMinLength	Verifies setting a title close to the minimum length.	Boundary Value Testing, Equivalence Testing

testSetTitle_WhenNormalLength	Verifies setting a title with a normal length.	Boundary Value Testing, Equivalence Testing
testSetTitle_WhenCloseToMaxLength	Verifies setting a title close to the maximum length.	Boundary Value Testing, Equivalence Testing
testSetTitle_WhenMoreThanMaxLength	Verifies that setting a title longer than the maximum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setPurchasePrice(double purchasePrice)
Description: Sets the purchase price of the book, ensuring it is non-negative and within a specified valid range.

Test Name	Description of Test	Test Type
testSetPurchasePrice_WithNegativePurchasePrice	Verifies that setting a negative purchase price throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetPurchasePrice_WhenPurchasePriceZero	Verifies setting a purchase price when it's zero.	Boundary Value Testing, Equivalence Testing
testSetPurchasePrice_WhenPurchasePriceValidValue	Verifies setting a valid purchase price.	Boundary Value Testing, Equivalence Testing
testSetPurchasePrice_WhenPurchasePriceMaxValue	Verifies setting the maximum valid purchase price.	Boundary Value Testing, Equivalence Testing
testSetPurchasePrice_WithMoreThanMaxPurchasePrice	Verifies that setting a purchase price higher than the maximum throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setOriginalPrice(double originalPrice)
Description: Sets the original price of the book, ensuring it is non-negative.

Test Name	Description of Test	Test Type
testSetOriginalPrice_WithNegativeOriginalPrice	Verifies that setting a negative original price throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetOriginalPrice_WhenOriginalPriceZero	Verifies setting an original price when it's zero.	Boundary Value Testing, Equivalence Testing
testSetOriginalPrice_WhenPurchasePriceValid	Verifies setting an original price when the purchase price is valid.	Boundary Value Testing, Equivalence Testing

Setter Method: void setSellPrice(double sellPrice)
Description: Sets the selling price of the book, ensuring it is greater than or equal to the purchase price.

Test Name	Description of Test	Test Type
testSetSellPrice_WithSellPriceSmallerThanPurchasePrice	Verifies that setting a sell price smaller than the purchase price throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetSellPrice_EqualWithPurchasePrice	Verifies setting a sell price equal to the purchase price.	Boundary Value Testing, Equivalence Testing

testSetSellPrice_BiggerThanPurchasePriceNormalValue	Verifies setting a sell price greater than the purchase price with a normal value.	Equivalence Testing
testSetSellPrice_ValidValueAGeneralCase	Verifies setting a valid sell price in a general case.	Boundary Value Testing, Equivalence Testing

Setter Method: void setAuthor(String author)
Description: Sets the author of the book, ensuring it is not null and does not exceed a specified maximum length.

Test Name	Description of Test	Test Type
testSetAuthor_WithNullSetAuthor	Verifies that setting the author to null throws an IllegalArgumentException.	Equivalence Testing
testSetAuthor_LessThanMinLength	Verifies that setting an author smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetAuthor_WhenMinimumValidLength	Verifies setting an author with the minimum valid length.	Boundary Value Testing, Equivalence Testing
testSetAuthor_WhenNormalLength	Verifies setting an author with a normal length.	Boundary Value Testing, Equivalence Testing
testSetAuthor_WhenCloseToMaxLength	Verifies setting an author close to the maximum length.	Boundary Value Testing, Equivalence Testing
testSetAuthor_WhenMoreThanMaxLength	Verifies that setting an author longer than the maximum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setSupplier(String supplier)
Description: Sets the supplier of the book, ensuring it is not null and does not exceed a specified maximum length.

Test Name	Description of Test	Test Type
testSetSupplier_WithNullValue	Verifies that setting the supplier to null throws an IllegalArgumentException.	Equivalence Testing
testSetSupplier_WhenLessThanLength	Verifies that setting a supplier smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetSupplier_WhenMinimumValidLength	Verifies setting a supplier with the minimum valid length.	Boundary Value Testing, Equivalence Testing
testSetSupplier_WhenNormalLength	Verifies setting a supplier with a normal length.	Boundary Value Testing, Equivalence Testing
testSetSupplier_WhenCloseToMaxLength	Verifies setting a supplier close to the maximum length.	Boundary Value Testing, Equivalence Testing
testSetSupplier_WhenMoreThanMaxLength	Verifies that setting a supplier longer than the maximum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setStock(int stock)
Description: Sets the stock quantity of the book, ensuring it is a valid positive value.

Test Name	Description of Test	Test Type
testSetStock_WithNegativeValue	Verifies that setting a negative stock value throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetStock_WhenTheValueOfItIsZero	Verifies that setting a stock value of zero throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetStock_WhenNormalValueBiggerThan0_UpperBound	Verifies setting a stock value of 1 (upper bound of normal range).	Boundary Value Testing, Equivalence Testing
testSetStock_WhenValidValueInRange	Verifies setting a stock value within a valid range (10).	Boundary Value Testing, Equivalence Testing

Setter Method: void setPurchaseDate(LocalDate purchaseDate)
Description: Sets the purchase date of the book, ensuring it is not null and not more than one year in the future.

Test Name	Description of Test	Test Type
testSetPurchaseDate_WithNullPurchaseDate	Verifies that setting the purchase date to null throws an IllegalArgumentException.	Equivalence Testing
testSetPurchaseDate_WhenMoreThanOneYearFromCurrentDate	Verifies that setting a purchase date more than one year from the current date throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetPurchaseDate_WhenExactlyOneYearFromCurrentDate	Verifies setting a purchase date exactly one year from the current date.	Boundary Value Testing, Equivalence Testing
testSetPurchaseDate_WhenDateIsTheLocalDateOfNow	Verifies setting a purchase date equal to the current date.	Boundary Value Testing, Equivalence Testing

Static Method: boolean canAddBook(Book selectedBook, int enteredQuantity)
Description: Determines if a book can be added based on its stock and the entered quantity.

Test Name	Description of Test	Test Type
testCanAddBook_WithStockZero	Verifies that canAddBook returns false when the stock is zero.	Boundary Value Testing, Equivalence Testing
testCanAddBook_WithEnteredQuantityZero	Verifies that canAddBook returns false when the entered quantity is zero.	Boundary Value Testing, Equivalence Testing
testCanAddBook_WithEnteredQuantityMoreThanStock	Verifies that canAddBook returns false when the entered quantity is more than the stock.	Boundary Value Testing, Equivalence Testing

testCanAddBook_WithEnteredQuantityEqualWithTheStock	Verifies that canAddBook returns true when the entered quantity is equal to the stock.	Boundary Value Testing, Equivalence Testing
testCanAddBook_WithEnteredQuantityLessThanTheStock	Verifies that canAddBook returns true when the entered quantity is less than the stock.	Boundary Value Testing, Equivalence Testing

Override Method: String toString()
Description: Returns a string representation of the book, including ISBN, title, prices, author, category, supplier, stock, and purchase date.

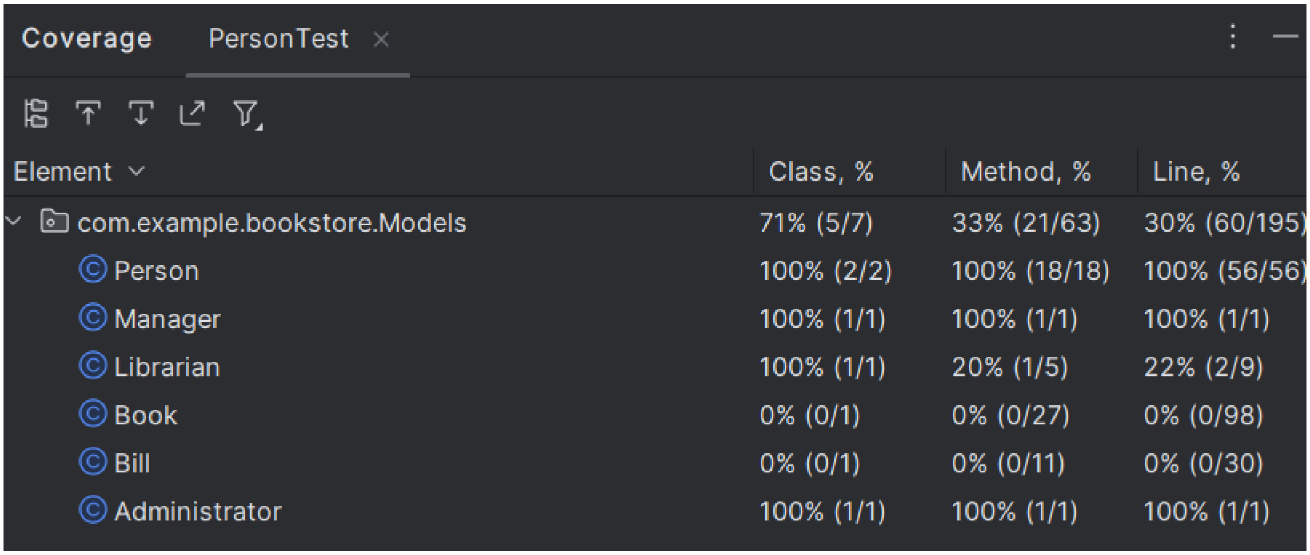
Test Name	Description of Test	Test Type
testToStringMethod	Verifies that the toString method generates the expected string representation of the Book object.	Equivalence Testing

Additional Method: String toStringBill()
Description: Returns a string representation of the book for use in printing bills, including ISBN, title, and author.

Test Name	Description of Test	Test Type
testToStringBillMethod	Verifies that the toStringBill method generates the expected string representation for billing purposes.	Equivalence Testing

Testing Person Model class

Coverage of the Person Model.



Coverage PersonTest

Element	Class, %	Method, %	Line, %
com.example.bookstore.Models	71% (5/7)	33% (21/63)	30% (60/195)
Person	100% (2/2)	100% (18/18)	100% (56/56)
Manager	100% (1/1)	100% (1/1)	100% (1/1)
Librarian	100% (1/1)	20% (1/5)	22% (2/9)
Book	0% (0/1)	0% (0/27)	0% (0/98)
Bill	0% (0/1)	0% (0/11)	0% (0/30)
Administrator	100% (1/1)	100% (1/1)	100% (1/1)

Testing the methods:

Setter Method: void setBirthday(String birthday)
Description: Sets the birthday of the person, ensuring it is not null or empty.

Test Name	Description of Test	Test Type
testSetBirthdayWhenItIsNull	Verifies that setting a null birthday throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetBirthdayWhenItIsEmpty	Verifies that setting an empty birthday throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetBirthdayWhenNormalLength	Verifies that setting a valid birthday updates the person's birthday.	Equivalence Testing

Setter Method: void setPhone(String phone)
Description: Sets the phone number of the person, ensuring it is not null and has a minimum length of 6.

Test Name	Description of Test	Test Type
testSetPhoneWhenItIsNull	Verifies that setting a null phone throws an IllegalArgumentException.	Equivalence Testing
testSetPhoneWhenLengthSmallerThan6	Verifies that setting a phone with a length smaller than 6 throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetPhoneWhenLengthIsEqualTo6	Verifies that setting a phone with a length equal to 6 updates the person's phone.	Boundary Value Testing, Equivalence Testing
testSetPhoneWhenLengthLargerThan6	Verifies that setting a phone with a length larger than 6 updates the person's phone.	Boundary Value Testing, Equivalence Testing
testSetPhoneWithEmptyString	Verifies that setting an empty phone string throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing

Setter Method: void setSalary(int salary)
Description: Sets the salary of the person, ensuring it is non-negative.

Test Name	Description of Test	Test Type
testSetSalaryWhenItIsNegative	Verifies that setting a negative salary throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetSalaryWhenItIsEqualTo0	Verifies that setting a salary equal to 0 updates the person's salary.	Boundary Value Testing, Equivalence Testing
testSetSalaryWhenItIsValidValue	Verifies that setting a valid salary updates the person's salary.	Boundary Value Testing, Equivalence Testing

Setter Method: void setUserName(String userName)
Description: Sets the username of the person, ensuring it is not null and has a length within the range of 5 to 15 characters.

Test Name	Description of Test	Test Type
testSetUserNameWhenItIsNull	Verifies that setting a null username throws an IllegalArgumentException.	Equivalence Testing
testSetUserNameWhenItIsSmallerThanMinLength	Verifies that setting a username smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetUserNameWhenItIsBiggerThanMaxLength	Verifies that setting a username larger than the maximum length	Boundary Value Testing, Equivalence Testing

	throws an IllegalArgumentException.	
testSetUserNameWhenLengthIsBetweenTheBounds	Verifies that setting a username with a valid length updates the person's username.	Boundary Value Testing, Equivalence Testing
testSetUserNameWhenLengthIsMaxLength	erifies that setting a username with the maximum length updates the person's username.	Boundary Value Testing
testSetUserNameWhenLengthIsMinLength	Verifies that setting a username with the minimum length updates the person's username.	Boundary Value Testing

Setter Method: void setPassword(String password)
Description: Sets the password of the person, ensuring it is not null and has a minimum length of 5 characters.

Test Name	Description of Test	Test Type
testSetPassword_WhenItIsSmallerThanMinLength	Verifies that setting a password smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetPasswordWhenLengthIsMinLength	Verifies that setting a password with the minimum length updates the person's password.	Boundary Value Testing
testSetPasswordWhenLengthIsBetweenTheBounds	Verifies that setting a password with a valid length updates the person's password.	Boundary Value Testing, Equivalence Testing

Setter Method: void setRole(Role role)
Description: Sets the role of the person, ensuring it is not null.

Test Name	Description of Test	Test Type
testSetRoleWhenItIsNull	Verifies that setting a null role throws an IllegalArgumentException.	Equivalence Testing
testSetRoleWhenItValidRoleThisExampleManager	Verifies that setting a valid role updates the person's role.	Boundary Value Testing, Equivalence Testing

Setter Method: void setName(String name)
Description: Sets the name of the person, ensuring it is not null and has a minimum length of 3 characters.

Test Name	Description of Test	Test Type
testSetNameWhenItIsNull	Verifies that setting a null name throws an IllegalArgumentException.	Equivalence Testing
testSetName_WhenItIsSmallerThanMinLength	Verifies that setting a name smaller than the minimum length throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetNameWhenItaLengthIsMinValue	Verifies that setting a name with the minimum length updates the person's name.	Boundary Value Testing
testSetNameWhenItaLengthIsOkay	Verifies that setting a name with a valid length updates the person's name.	Boundary Value Testing, Equivalence Testing

Static Method: Person createPerson(String name, String username, String password, String birthday, int salary, String phone, Role role)
Description: Creates and returns a Person object based on the provided parameters, including a switch statement to handle different roles.

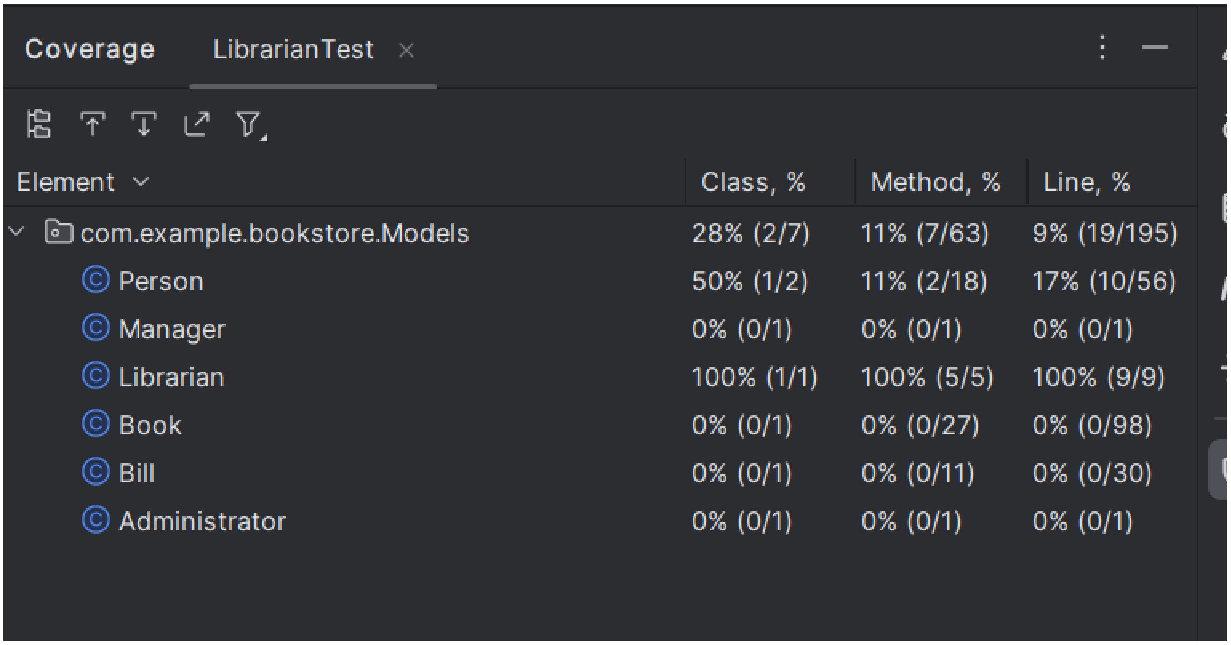
Test Name	Description of Test	Test Type
testCreatePerson_WhenRoleIsNotOneOfTheCases	Verifies that creating a person with an invalid role throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testCreatePersonWhenRoleIsLibrarian	Verifies that creating a person with the role of Librarian returns a Librarian instance.	Boundary Value Testing, Equivalence Testing
testCreatePersonWhenRoleIsManager	Verifies that creating a person with the role of Manager returns a Manager instance.	Boundary Value Testing, Equivalence Testing
testCreatePersonWhenRoleIsAdministrator	Verifies that creating a person with the role of Administrator returns an Administrator instance.	Boundary Value Testing, Equivalence Testing

Override Method: String toString()
Description: Returns a string representation of the person, including name, birthday, phone, salary, username, password, and role.

Test Name	Description of Test	Test Type
testPersonToStringMethod	Verifies that the toString method generates the expected string representation of the Person object.	Equivalence Testing

Testing Librarian Model class

Coverage of the Librarian Model.



Element	Class, %	Method, %	Line, %
com.example.bookstore.Models	28% (2/7)	11% (7/63)	9% (19/195)
Person	50% (1/2)	11% (2/18)	17% (10/56)
Manager	0% (0/1)	0% (0/1)	0% (0/1)
Librarian	100% (1/1)	100% (5/5)	100% (9/9)
Book	0% (0/1)	0% (0/27)	0% (0/98)
Bill	0% (0/1)	0% (0/11)	0% (0/30)
Administrator	0% (0/1)	0% (0/1)	0% (0/1)

Testing the methods:

Setter Method: void setTotalBilled(double totalBilled)
Description: Sets the total amount billed by the librarian, ensuring it is non-negative; throws IllegalArgumentException if the value is negative.

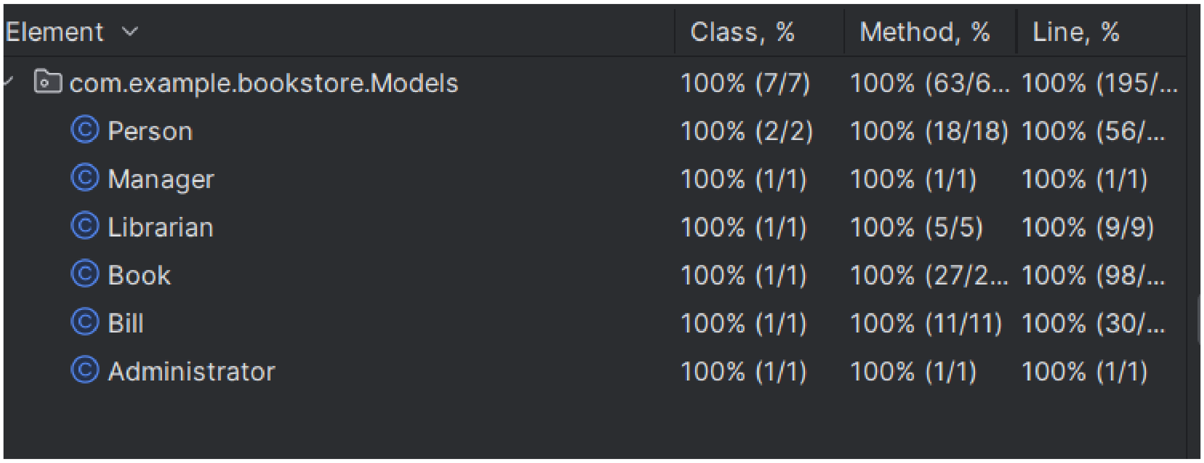
Test Name	Description of Test	Test Type
testSetTotalBilledNegativeValue	Verifies that setting a negative total billed value throws an IllegalArgumentException.	Boundary Value Testing, Equivalence Testing
testSetTotalBilledValueJustMinValue	Verifies that setting the total billed value to the minimum value updates the librarian's total billed value.	Boundary Value Testing, Equivalence Testing

testSetTotalBilledValuePositive	Verifies that setting a positive total billed value updates the librarian's total billed value.	Boundary Value Testing, Equivalence Testing
testTheConstructorWithoutTotalBilledToStringMethod	Verifies that the toString method generates the expected string representation of the Librarian object without a total billed value. Test Type: Equivalence Testing	

Override Method: String toString()
Description: Returns a string representation of the Librarian object, including the values obtained from the toString method of the super class (Person) and appending the totalBilled value.

Test Name	Description of Test	Test Type
testPersonToStringMethod	Verifies that the toString method generates the expected string representation of the Librarian object with a total billed value.	Equivalence Testing

Total Coverage of the Models.

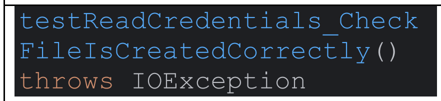
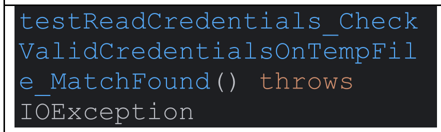
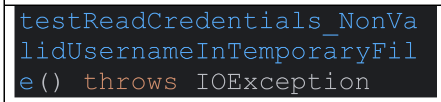
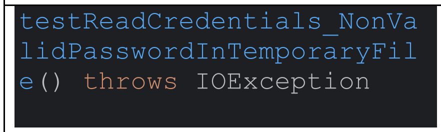
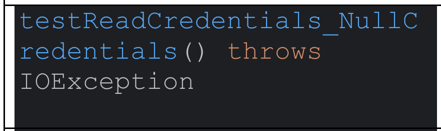
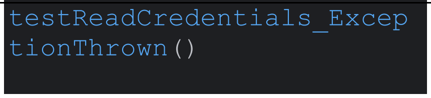


Element	Class, %	Method, %	Line, %
com.example.bookstore.Models	100% (7/7)	100% (63/6...	100% (195/...
Person	100% (2/2)	100% (18/18)	100% (56/...
Manager	100% (1/1)	100% (1/1)	100% (1/1)
Librarian	100% (1/1)	100% (5/5)	100% (9/9)
Book	100% (1/1)	100% (27/2...	100% (98/...
Bill	100% (1/1)	100% (11/11)	100% (30/...
Administrator	100% (1/1)	100% (1/1)	100% (1/1)

Testing WritingToFile class

UNIT TESTING

Method: String readCredentials(String username, String password, String filepath)
Description: The readCredentials method validates user credentials by checking them against a file. It takes a username, password, and filepath as parameters, reads the file line by line, and returns the corresponding role if there's a match. It returns null if no match is found or an exception occurs during file reading. It is designed for a basic user authentication system with roles stored in a file.

Test Name	Description of Test	Test Type
 `testReadCredentials_CheckFileIsCreatedCorrectly() throws IOException`	The test is focused on ensuring that the method, when given a file path (using temporary file), reads credentials correctly and creates a non-empty file.	Unit Testing
 `testReadCredentials_CheckValidCredentialsOnTempFile_MatchFound() throws IOException`	This Java test method is designed to validate the functionality of the method by checking whether it correctly reads credentials from a temporary file and returns the expected role for a given username and password combination.	Unit Testing
 `testReadCredentials_NonValidUsernameInTemporaryFile() throws IOException`	This Java test method is designed to verify the behavior of the method when provided with a non-valid username (one that does not exist in the temporary file).	Unit Testing
 `testReadCredentials_NonValidPasswordInTemporaryFile() throws IOException`	This test method appears to be designed to check the behavior of the method when a non-valid password is provided for a specific username in the temporary file.	Unit Testing
`testReadCredentials_EmptyFile() throws IOException`	This test method is designed to verify the behavior of the readCredentials method when the temporary file is empty	Unit Testing
 `testReadCredentials_NullCredentials() throws IOException`	This test method is designed to verify the behavior of the readCredentials method when null credentials (username and password) are provided	Unit Testing
 `testReadCredentials_ExceptionThrown()`	This test method is designed to verify the behavior of the readCredentials method when an exception is thrown during its execution.	Unit Testing

Method: void writeRoles(String filePath, ArrayList<Person> peopleList)
Description: The writeRoles method initializes or updates a file with user roles. It appends administrator credentials as the first line and iterates through a list of people, adding their username, password, and role on separate lines. Any exceptions during the process result in a runtime exception.

Test Name	Description of Test	Test Type
`testWriteRoles_CheckFileIsCreatedCorrectly() throws IOException`	The test method creates a temporary file, invokes the writeRoles method with an empty list of persons, and asserts that the temporary file exists and is not empty.	Unit Testing
`testWriteRoles_CheckContentOfFileWhenYouDontWriteAnything() throws IOException`	This test method, aims to verify the behavior of the writeRoles method when an empty list of persons is provided.	Unit Testing
`testWriteRoles_CheckContentOfFile() throws IOException`	This test method aims to verify the behavior of the writeRoles method when a list of persons is provided.	Unit Testing
`testWriteRoles_ThrowException()`	This test method, is designed to check the behavior of the writeRoles method when it is called with null values for the file path and the list of persons.	Unit Testing

Method: ObservableList<Book> getBooks(String filepath)
Description: The getBooks method retrieves book data from a specified file, creating an ObservableList of Book objects. It reads the file, processes each line into data fields, and constructs Book objects accordingly. Any exceptions during the file reading result in a runtime exception.

Test Name	Description of Test	Test Type
`testGetBooks_CheckFileIsCreatedCorrectlyButIsEmpty() throws IOException`	This test method, tests the behavior of the getBooks method. This test ensures that when the getBooks method is called on a file that exists but is empty, it returns an empty list of books.	Unit Testing
`testGetBooks_ObservableListNotNull() throws IOException`	This test ensures that when the getBooks method is called on a file containing book entries, it returns a non-null ObservableList of books.	Unit Testing
`testGetBooks_ContainsTheCorrectNumberOfBooks() throws IOException`	This test ensures that the getBooks method correctly reads and returns the expected number of books when called on a file with entries.	Unit Testing

`testGetBooks_AttributesOfTheBookCorrectlyPopulated() throws IOException`	This test ensures that the getBooks method correctly populates the attributes of the first book when called on a file with book entries.	Unit Testing
`testGetBooks_ThrowException()`	This test ensures that the getBooks method handles the case where a null file path is provided and throws a RuntimeException as expected.	Unit Testing

Method: ObservableList<Person> getPersons(String filepath)
Description: The getPersons method retrieves person data from a specified file, creating an ObservableList of Person objects. It reads the file, processes each line into data fields, and constructs Person objects based on the role field. Any exceptions during the file reading result in a runtime exception.

Test Name	Description of Test	Test Type
`testGetPersons_CheckFileIsCreatedCorrectlyButIsEmpty() throws IOException`	This test is checking if the getPersons method behaves correctly when reading from a file that exists but has no content. It ensures that the method handles an empty file by returning an empty list. The use of a temporary file allows the test to be isolated and independent of external factors.	Unit Testing
`testGetPersons_ObservableListNotNull() throws IOException`	The test ensures that the getPersons method functions correctly in the presence of valid data in the file, and it verifies that the returned ObservableList is not null, indicating successful data retrieval.	Unit Testing
`testGetPersons_ContainsTheCorrectNumberOfPersons() throws IOException`	The test ensures that the getPersons method functions correctly in terms of returning the expected number of persons when the file contains valid data. It provides a basic check on the correctness of the method's behavior.	Unit Testing
`testGetPersons_AttributesOfTheBookCorrectlyPopulated() throws IOException`	The test provides a specific and detailed check on the correctness of the getPersons method by verifying the attributes of a Person object. It helps ensure that the method correctly parses and populates the attributes from the file content.	Unit Testing
`testGetPersons_ThrowException()`	The test focuses on verifying the expected exception behavior of the getPersons method when encountering an invalid input (in this case, a null file path). It ensures that the method reacts appropriately to such situations by throwing the expected exception.	Unit Testing

Method: String getNumberOfLibrarians(ArrayList<Person> peopleList)
Description: The getNumberOfLibrarians method counts the number of librarians in the provided list of people. It iterates through each person, checks if they are an instance of the Librarian class, and increments the count accordingly. The final count is returned as a string.

Test Name	Description of Test	Test Type
`testGetNumberOfLibrarians_inWritingToFilesButIsEmptyList()`	This test case is specifically designed to verify the behavior of the method when provided with an empty list of people, confirming that it correctly returns "0" in such a scenario.	Unit Testing
`testGetNumberOfLibrarians_inWritingToFiles()`	This test case is focused on verifying that the getNumberOfLibrarians method behaves correctly when provided with a list of people that includes at least one librarian. The expected result is "1" to indicate the correct counting of librarians.	Unit Testing

Method: String getNumberOfManagers(ArrayList<Person> peopleList)
Description: The getNumberOfManagers method counts the number of managers in the provided list of people. It iterates through each person, checks if they are an instance of the Manager class, and increments the count accordingly. The final count is returned as a string.

Test Name	Description of Test	Test Type
`testGetNumberOfManagers_inWritingToFilesButIsEmptyList()`	This test checks if the getNumberOfManagers method correctly returns "0" when the input list of people is empty.	Unit Testing
`testGetNumberOfManagers_inWritingToFiles()`	This test checks if the getNumberOfManagers method correctly counts the number of managers in a list of people that includes at least one manager.	Unit Testing

Method: int getNumberOfBills(String Filepath)
Description: The getNumberOfBills method takes a file path as input, creates a File object representing the directory, and checks if the directory exists. If the directory exists, the method returns the number of files (bills) in that directory. In case of a NullPointerException, it throws a RuntimeException. If the directory does not exist, it returns 0.

Test Name	Description of Test	Test Type
`testGetNumberOfBills_WhenWePassAFileNotADirectory_ThrowsException() throws IOException`	This test checks if calling getNumberOfBills with a file that is not a directory throws a RuntimeException.	Unit Testing
`testGetNumberOfBills_InTheNONExistingDirectory()`	This test checks if calling getNumberOfBills with a path to a non-existing directory returns 0.	Unit Testing
`testGetNumberOfBills_InTheExistingDirectoryIfBiggerThen0ThenCorrectNumberOfFiles()`	This test checks if calling getNumberOfBills with the path to an existing directory returns a number greater than 0.	Unit Testing

Method: void writeBill(String billId, double totalBill, List<Book> books, String filePath)
Description: The writeBill method takes the bill ID, total bill amount, a list of books, and a file path as input. It creates a File object representing the bill file and obtains the current date and time. The method then creates a new file and uses a FileWriter to write the bill header information, including the bill ID and date. It iterates over the list of books, appending each book's details to the file. Finally, it writes the total bill amount and closes the FileWriter. If an exception occurs during the process, it throws a RuntimeException with the corresponding error message.

Test Name	Description of Test	Test Type
`testWriteBills_CheckFileIsCreatedCorrectly()`	This test checks if the writeBill method correctly creates a bill file.	Unit Testing
`testWriteBills_CheckFileContentIsCreatedCorrectly()`	This test checks if the writeBill method correctly writes the content of the bill file.	Unit Testing
`testWriteBills_ThrowException()`	This test checks if the writeBill method throws a RuntimeException when provided with invalid input.	Unit Testing

Method: double getTotalBill(String filepath)
Description: The getTotalBill method checks if the file at the given filepath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method. If successful, it returns the read value as the total bill amount. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
`testGetTotalBillWithMockExistingFile()`	Verifies if getTotalBill returns the correct total bill from a mock file with existing content.	Unit Tests Mock
`testGetTotalBillWithMockNonExistingFile()`	Verifies if getTotalBill returns 0 when attempting to read from a mock non-existing file.	Unit Tests Mock
`testGetTotalBillWithMockIOException()`	Verifies if getTotalBill handles an IOException and returns 0.	Unit Tests Mock

Method: double getTotalCost(String filePath)
Description: The getTotalCost method checks if the file at the given filePath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method. If successful, it returns the read value as the total cost. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
testGetTotalCostWithValidFile()	Verifies if getTotalCost returns the correct total cost from a mock file with existing content.	Unit Tests Mock
testGetTotalCostWithNonExistingFile()	Verifies if getTotalCost returns 0 when attempting to read from a mock non-existing file.	Unit Tests Mock
testGetTotalCostWithIOException()	Verifies if getTotalCost handles an IOException and returns 0.	Unit Tests Mock

Method: int getBooksSold(String filePath)
Description: The getBooksSold method checks if the file at the given filePath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method and then casts it to an integer. The method returns this integer value as the number of books sold. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
testGetBooksSoldWithExistingFileMocking()	This test case checks the getBooksSold method when it is provided with the path to an existing file. It sets up a mock file operation with a flag indicating that the file exists and a mock value of 123. The test then asserts that the method returns the expected value of 123.	Unit Tests Mock
testGetBooksSoldWithNonExistingFileMocking()	This test case checks the getBooksSold method when it is provided with the path to a non-existing file. It sets up a mock file operation with a flag indicating that the file does not exist and a mock value of 0.0. The test then asserts that the method returns the expected value of 0.	Unit Tests Mock
testGetBooksSoldWithIOExceptionMocking()	This test case checks the getBooksSold method when an IOException occurs during file reading. It sets up a mock file operation with a flag indicating that the file exists, a mock value of 0.0, and overrides the readDoubleFromFile method to throw an IOException. The test then asserts that the method returns the expected value of 0 in the presence of an IOException.	Unit Tests Mock

Method: void writeBooks(String filepath, ArrayList<Book> bookArrayList)
Description: Within a try-catch block, the method creates a File object representing the specified file path. It then creates a FileWriter instance, writer, to write data to the file. The method iterates over each Book object in the bookArrayList and writes its string representation (obtained using the toString method) followed by a newline character to separate entries. After writing all book information, the FileWriter is closed to save the changes. If an exception occurs during the process (e.g., IOException), a runtime exception is thrown with an error message.

Test Name	Description of Test	Test Type
`testWriteBooks_CheckFileIsCreatedCorrectly() throws IOException`	Ensures that the writeBooks method successfully creates a file when provided with valid book data.	Unit Tests
`testWriteBooks_CheckFileContentIsCreatedCorrectly() throws IOException`	Validates that the content written to the file by the writeBooks method matches the expected format.	Unit Tests
`testWriteBooks_ThrowException()`	Ensures that calling the writeBooks method with null parameters results in a RuntimeException.	Unit Tests

Method: void writePersons(String filepath, ArrayList<Person> personArrayList)
Description: The writePersons method efficiently writes a list of persons to a specified file. It employs a streamlined structure, featuring try-with-resources for resource management. In case of an error, a concise runtime exception is thrown with a clear message.

Test Name	Description of Test	Test Type
`testWritePersons_CheckFileIsCreated() throws IOException`	Verifies that the writePersons method creates a file when provided with a valid list of Person objects.	Unit Tests
`testWritePersons_CheckFileContentIsCreatedCorrectly() throws IOException`	Validates that the content written to the file by the writePersons method matches the expected format.	Unit Tests
`testWritePersons_ThrowException()`	Ensures that calling the writePersons method with null parameters results in a RuntimeException.	Unit Tests

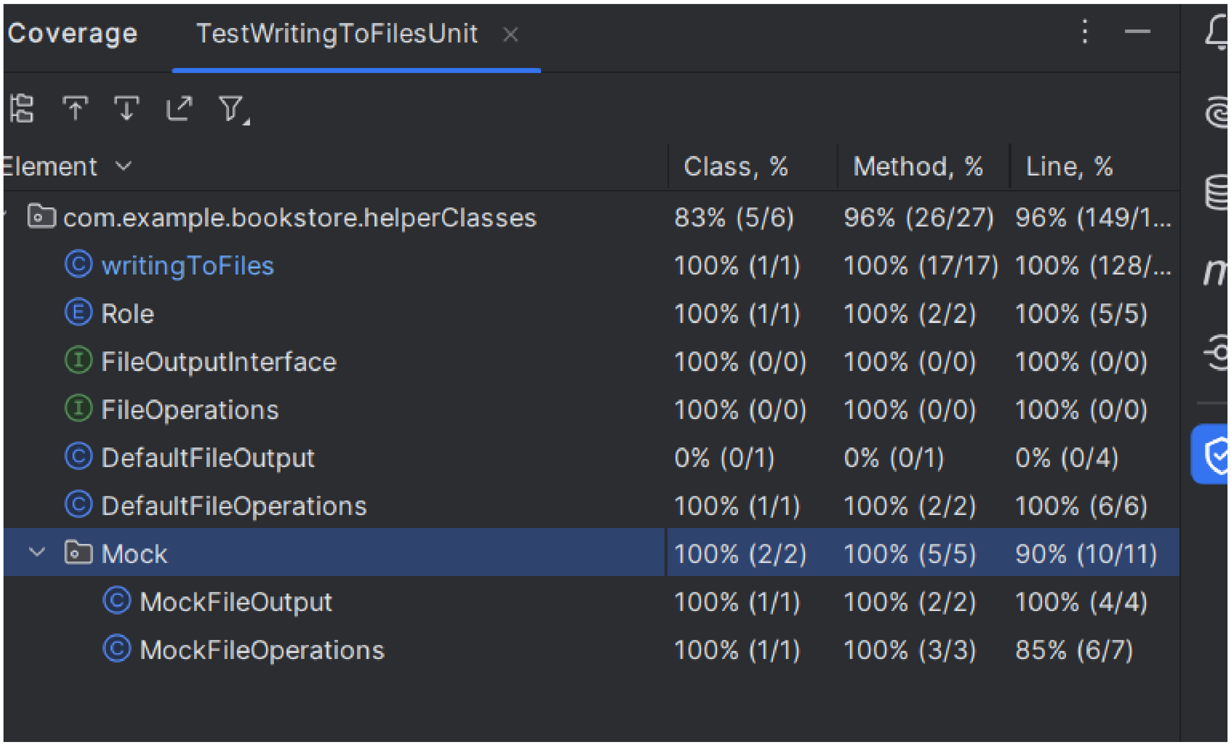
Method: void writeTotalBill(double total, String filePath, FileOutputInterface fileOutput)
Description: The writeTotalBill method simplifies the process of writing a total bill to a file using a provided FileOutputInterface. It delegates the task to the interface's writeDoubleToFile method, handling any potential IOException by throwing a clear and concise runtime exception with the error message.

Test Name	Description of Test	Test Type
`testWriteTotalBillSuccessMock()`	Verifies that the writeTotalBill method does not throw an exception when provided with a valid total, file path, and a mock FileOutputInterface that does not throw an exception during write.	Unit Tests
`testWriteTotalBillWithExceptionMock()`	Validates that the writeTotalBill method correctly handles an exception thrown by the mock FileOutputInterface.	Unit Tests

Method: void writeTotalCost(double total, String filepath, FileOutputInterface fileOutput)
Description: The writeTotalCost method simplifies the process of writing a total cost to a file using a provided FileOutputInterface. It delegates the task to the interface's writeDoubleToFile method, handling any potential IOException by throwing a clear and concise runtime exception with the error message.

Test Name	Description of Test	Test Type
`testWriteTotalCostSuccessMock()`	Verifies that the writeTotalCost method does not throw an exception when provided with a valid total, file path, and a mock FileOutputInterface that does not throw an exception during write.	Unit Tests
`testWriteTotalCostWithExceptionMock()`	Validates that the writeTotalCost method correctly handles an exception thrown by the mock FileOutputInterface.	Unit Tests

Coverage of the Unit Testing.



Element	Class, %	Method, %	Line, %
com.example.bookstore.helperClasses	83% (5/6)	96% (26/27)	96% (149/1...
writingToFiles	100% (1/1)	100% (17/17)	100% (128/...
Role	100% (1/1)	100% (2/2)	100% (5/5)
FileOutputInterface	100% (0/0)	100% (0/0)	100% (0/0)
FileOperations	100% (0/0)	100% (0/0)	100% (0/0)
DefaultFileOutput	0% (0/1)	0% (0/1)	0% (0/4)
DefaultFileOperations	100% (1/1)	100% (2/2)	100% (6/6)
Mock	100% (2/2)	100% (5/5)	90% (10/11)
MockFileOutput	100% (1/1)	100% (2/2)	100% (4/4)
MockFileOperations	100% (1/1)	100% (3/3)	85% (6/7)

INTEGRATION TESTING

Method: double getTotalBill(String filepath)
Description: The getTotalBill method checks if the file at the given filepath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method. If successful, it returns the read value as the total bill amount. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
`testGetTotalBillWithExistingTempFile() throws IOException`	Verifies if getTotalBill correctly reads the total bill from an existing temporary file.	Integration Test
`testGetTotalBillWithNONExistingTempFile()`	Verifies if getTotalBill returns 0 when attempting to read from a non-existing temporary file.	Integration Test
`testGetTotalBillWithExistingTempFileButEmpty() throws IOException`	Verifies if getTotalBill returns 0 when attempting to read from an existing but empty temporary file.	Integration Test

Method: double getTotalCost(String filePath)

Description: The getTotalCost method checks if the file at the given filePath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method. If successful, it returns the read value as the total cost. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
`testGetTotalCostWithValidTempFile() throws IOException`	Verifies if getTotalCost correctly reads the total cost from an existing temporary file.	Integration Test
`testGetTotalCostWithNONExistingTempFile()`	Verifies if getTotalCost returns 0 when attempting to read from a non-existing temporary file.	Integration Test
`testGetTotalCostWithExistingTempFileButEmpty() throws IOException`	Verifies if getTotalCost returns 0 when attempting to read from an existing but empty temporary file.	Integration Test

Method: int getBooksSold(String filePath)

Description: The getBooksSold method checks if the file at the given filePath exists using the fileExists method from fileOperations. If the file exists, it attempts to read a double value from the file using the readDoubleFromFile method and then casts it to an integer. The method returns this integer value as the number of books sold. If an IOException occurs during the process, it returns 0. If the file does not exist, it also returns 0.

Test Name	Description of Test	Test Type
`testGetBooksSoldWithExistingTempFile() throws IOException`	Verifies that the getBooksSold method correctly retrieves the total books sold from an existing temporary file.	Integration Test
`testGetTotalBooksSoldWithNONExistingTempFile()`	Verifies that the getBooksSold method returns 0 when provided with the path of a non-existing file.	Integration Test

`testGetTotalBooksWithExistingTempFileButEmpty() throws IOException`	Verifies that the getBooksSold method correctly handles an existing but empty temporary file by returning 0.	Integration Test

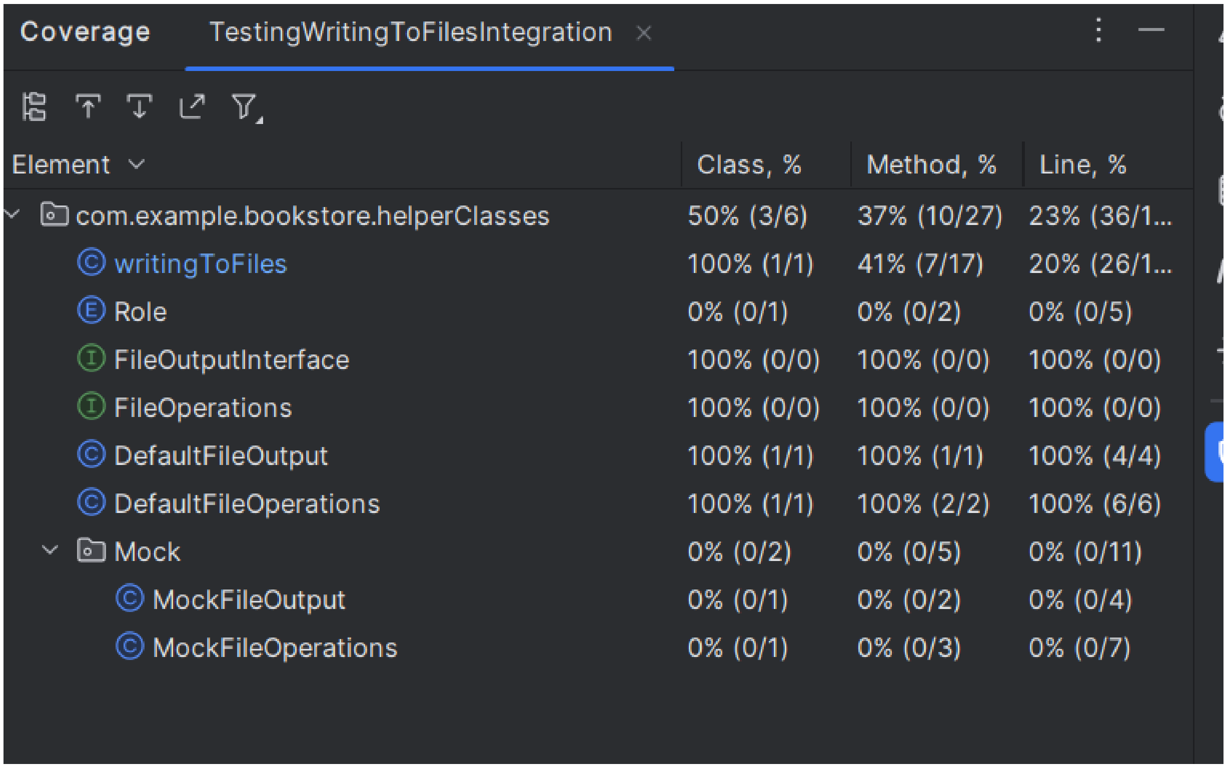
Method: void writeTotalBill(double total, String filePath, FileOutputInterface fileOutput)
Description: The writeTotalBill method simplifies the process of writing a total bill to a file using a provided FileOutputInterface. It delegates the task to the interface's writeDoubleToFile method, handling any potential IOException by throwing a clear and concise runtime exception with the error message.

Test Name	Description of Test	Test Type
`testWriteTotalBillSuccessWithTempFile() throws IOException`	Verifies that the writeTotalBill method successfully writes the total bill to a temporary file using the DefaultFileOutput.	Integration Test
`testWriteTotalBillThrowExceptionWithTempFile()`	Verifies that the writeTotalBill method correctly handles an exception (IOException) when attempting to write the total bill to a non-writable file.	Integration Test

Method: void writeTotalCost(double total, String filepath, FileOutputInterface fileOutput)
Description: The writeTotalCost method simplifies the process of writing a total cost to a file using a provided FileOutputInterface. It delegates the task to the interface's writeDoubleToFile method, handling any potential IOException by throwing a clear and concise runtime exception with the error message.

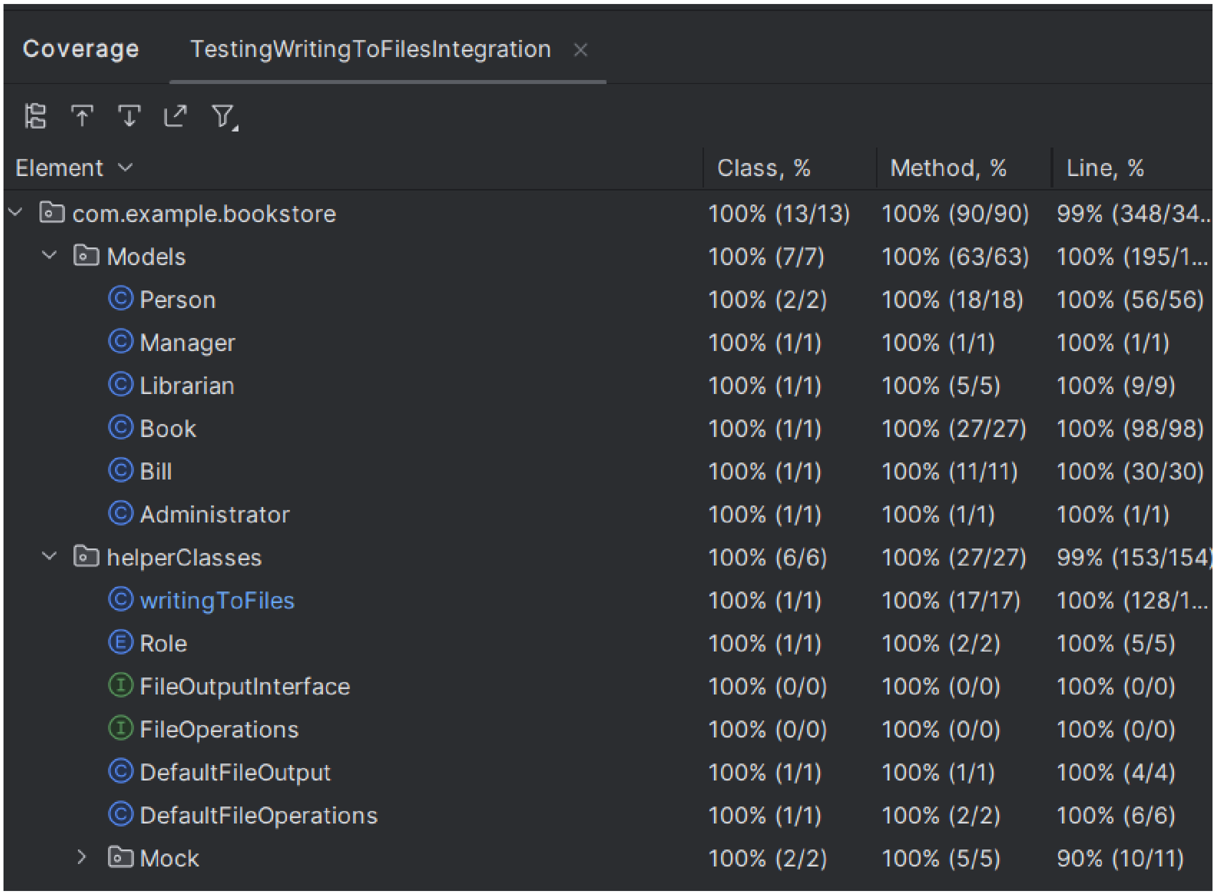
Test Name	Description of Test	Test Type
`testWriteTotalCostSuccessWithTempFile() throws IOException`	Verifies that the writeTotalCost method successfully writes the total cost to a temporary file using the DefaultFileOutput.	Integration Test
`testWriteTotalCostThrowExceptionWithTempFile()`	Verifies that the writeTotalCost method correctly handles an exception (IOException) when attempting to write the total cost to a non-writable file.	Integration Test

Coverage of the Integration Testing.



Element	Class, %	Method, %	Line, %
com.example.bookstore.helperClasses	50% (3/6)	37% (10/27)	23% (36/1…
writingToFiles	100% (1/1)	41% (7/17)	20% (26/1…
Role	0% (0/1)	0% (0/2)	0% (0/5)
FileOutputInterface	100% (0/0)	100% (0/0)	100% (0/0)
FileOperations	100% (0/0)	100% (0/0)	100% (0/0)
DefaultFileOutput	100% (1/1)	100% (1/1)	100% (4/4)
DefaultFileOperations	100% (1/1)	100% (2/2)	100% (6/6)
Mock	0% (0/2)	0% (0/5)	0% (0/11)
MockFileOutput	0% (0/1)	0% (0/2)	0% (0/4)
MockFileOperations	0% (0/1)	0% (0/3)	0% (0/7)

Total Coverage of writing to files and models.



Element	Class, %	Method, %	Line, %
com.example.bookstore	100% (13/13)	100% (90/90)	99% (348/34…
Models	100% (7/7)	100% (63/63)	100% (195/1…
Person	100% (2/2)	100% (18/18)	100% (56/56)
Manager	100% (1/1)	100% (1/1)	100% (1/1)
Librarian	100% (1/1)	100% (5/5)	100% (9/9)
Book	100% (1/1)	100% (27/27)	100% (98/98)
Bill	100% (1/1)	100% (11/11)	100% (30/30)
Administrator	100% (1/1)	100% (1/1)	100% (1/1)
helperClasses	100% (6/6)	100% (27/27)	99% (153/154)
writingToFiles	100% (1/1)	100% (17/17)	100% (128/1…
Role	100% (1/1)	100% (2/2)	100% (5/5)
FileOutputInterface	100% (0/0)	100% (0/0)	100% (0/0)
FileOperations	100% (0/0)	100% (0/0)	100% (0/0)
DefaultFileOutput	100% (1/1)	100% (1/1)	100% (4/4)
DefaultFileOperations	100% (1/1)	100% (2/2)	100% (6/6)
Mock	100% (2/2)	100% (5/5)	90% (10/11)

Besis Path testing for a method of my choice. (GET PERSONS AT WRITETOFILE)

Label the different statements and branches:

```
S1: ObservableList<Person> people = FXCollections.observableArrayList();
S2: try {
S3: File file = new File(filepath);
S4: if (file.exists()) {
S5: Scanner scanner = new Scanner(file);
S6: while (scanner.hasNextLine()) {
S7: String[] Data = scanner.nextLine().split(",");
S8: if (Data[6].equalsIgnoreCase("Librarian")) {
S9: people.add(new Librarian(Data[0], Data[4], Data[5], Data[1], Integer.parseInt(Data[3]), Data[2], Role.Librarian, Double.parseDouble(Data[7])));
S10: } else if (Data[6].equalsIgnoreCase("Manager")) {
S11: people.add(new Manager(Data[0], Data[4], Data[5], Data[1], Integer.parseInt(Data[3]), Data[2], Role.Manager));
S12: } else if (Data[6].equalsIgnoreCase("Administrator")) {
S13: people.add(new Administrator(Data[0], Data[4], Data[5], Data[1], Integer.parseInt(Data[3]), Data[2], Role.Administrator));
S14: }   S15: }
S16: } (end of while loop)
S17: } (end of if statement)
S18: } catch (Exception e) {
S19: throw new RuntimeException(e.getMessage());
S20: } (end of try block)
S21: return people;
```

Now, let's identify the different paths through the code:

Test Case 1: Existing File with Librarian
Path: S2 → S3 → S4 (True) → S5 → S6 → S7 → S8 → S9 → S15 → S16 → S18 → S20 → S21

Test Case 2: Existing File with Manager
Path: S2 → S3 → S4 (True) → S5 → S6 → S7 → S8 → S10 → S11 → S15 → S16 → S18 → S20 → S21

Test Case 3: Existing File with Administrator
Path: S2 → S3 → S4 (True) → S5 → S6 → S7 → S8 → S12 → S13 → S15 → S16 → S18 → S20 → S21

Test Case 4: Non-existing File
Path: S2 → S3 → S4 (False) → S21

Test Case 5: Existing File with Unknown Role
Path: S2 → S3 → S4 (True) → S5 → S6 → S7 → S8 → S15 → S16 → S18 → S20 → S21

Test Case 6: Empty File
Path: S2 → S3 → S4 (True) → S5 → S6 → S21

THE CODE AT
src/test/java/com/example/bookstore/helperClasses/BasisPathTestingForGetPersons.java

Main Application Test

Thread 1: Test Null Login ()

The purpose of this unit test is to verify the system's behavior when a user attempts to log in without entering any credentials (null login). The focus is on ensuring that the system appropriately responds to such scenarios and displays an alert dialog with a visible "OK" button, indicative of an error message.

Success Scenario:

The test passes if the system displays an Alert dialog with a visible "OK" button in response to a null login attempt.

Failure Scenario:

The test fails if the "OK" button is not visible, indicating a deviation from the expected behavior.

Thread 2: Test Null Role ()

The purpose of this unit test is to verify the system's behavior when a user with invalid credentials attempts to log in. Specifically, it simulates a scenario where both the username and password are invalid, and it checks if the system displays an appropriate error message, possibly in an Alert dialog.

Success Scenario:

The test passes if the system displays an error message (possibly in an Alert dialog) with a visible "OK" button, indicating a failed login attempt due to invalid credentials.

Failure Scenario:

The test fails if the "OK" button is not visible, signaling a deviation from the expected behavior.

Thread 3: Test Valid Credentials Administrator ()

The purpose of this unit test is to verify the system's behavior when a user with valid administrator credentials attempts to log in. It simulates the scenario where the username is "admin," the password is "admin," and checks if the system successfully grants access to the administrator panel.

Success Scenario:

The test passes if the system successfully logs in the user with the "admin" username and "admin" password and displays the administrator panel.

Failure Scenario:

The test fails if the administrator panel is not visible after the login attempt, indicating a deviation from the expected behavior.

Thread 4: Test Valid Credentials Librarian ()

The purpose of this unit test is to verify the system's behavior when a user with valid librarian credentials attempts to log in. It simulates the scenario where the username is "Llaca," the password is "12345," and checks if the system successfully grants access to the librarian panel.

Success Scenario:

The test passes if the system successfully logs in the user with the "Llaca" username and "12345" password and displays the librarian panel.

Failure Scenario:

The test fails if the librarian panel is not visible after the login attempt, indicating a deviation from the expected behavior.

Thread 5: Test Valid Credentials Manager ()

The purpose of this unit test is to verify the system's behavior when a user with valid manager credentials attempts to log in. It simulates the scenario where the username is "1," the password is "12345678," and checks if the system successfully grants access to the manager panel.

Success Scenario:

The test passes if the system successfully logs in the user with the "1" username and "12345678" password and displays the manager panel.

Failure Scenario:

The test fails if the manager panel is not visible after the login attempt, indicating a deviation from the expected behavior.

Admin Panel Test (these threads happen when you log in with administrator Credentials)

Thread 6: Test Open Employee Register Success ()

The purpose of this unit test is to ensure that the system allows an administrator to successfully open the employee registration form, enter employee details, and register a new employee.

Success Scenario:

The test passes if the system successfully registers the new employee, and the confirmation message is visible.

Failure Scenario:

The test fails if the confirmation message is not visible or if any step of the registration process encounters an issue.

Thread 7: Test Open Employee Register Empty Credentials ()

The purpose of this unit test is to ensure that the system handles the scenario where an administrator attempts to register a new employee with empty credentials. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects empty credentials and displays the expected error message.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 8: Test Open Employee Register Exception Thrown ()

The purpose of this unit test is to ensure that the system appropriately handles exceptions thrown during the attempt to register a new employee by an administrator. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects the exception (in this case, due to a non-numeric salary) and displays the expected error message with an "OK" button.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 9: Test Add New Book Success ()

The purpose of this unit test is to ensure that the system allows an administrator to successfully add a new book. The test checks whether the system correctly handles the input of book details, submits the information, and displays a success message.

Success Scenario:

The test passes if the system successfully adds the new book, and the confirmation message is visible.

Failure Scenario:

The test fails if the confirmation message is not visible or if any step of the process encounters an issue.

Thread 10: Test Add New Book Empty Credentials ()

The purpose of this unit test is to ensure that the system handles the scenario where an administrator attempts to add a new book with empty credentials or fields. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects empty credentials and displays the expected error message.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 11: Test Add New Book Exception Thrown ()

The purpose of this unit test is to ensure that the system appropriately handles exceptions thrown during the attempt to add a new book by an administrator. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects the exception (in this case, due to a non-numeric purchased price) and displays the expected error message with an "OK" button.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 12: Test Open Employee List Success Employee Registration Updated ()

The purpose of this unit test is to ensure that the system updates the employee list successfully after a new employee is registered. The test checks whether the system correctly displays the newly registered employee in the employee list.

Success Scenario:

The test passes if the employee list is successfully updated, and it is not empty after registering a new employee.

Failure Scenario:

The test fails if the employee list is empty or if any step of the process encounters an issue.

Thread 13: Test Open Employee List After Error In Employee Registration ()

The purpose of this unit test is to ensure that the system handles errors appropriately when an administrator attempts to add a new employee with invalid data, and subsequently, the employee list remains unchanged. The test checks whether the system displays an error message and confirms that the employee list is not updated after encountering an error in registration.

Success Scenario:

The test passes if the system displays an error message and the employee list remains unchanged after attempting to register an employee with an error.

Failure Scenario:

The test fails if the employee list is updated or if any step of the process encounters an issue.

Thread 14: Test Open Manage Books List Success Updated After Book Addition ()

The purpose of this unit test is to ensure that the system updates the book management list successfully after a new book is added. The test checks whether the system correctly displays the newly added book in the book management list.

Success Scenario:

The test passes if the book management list is successfully updated, and it is not empty after adding a new book.

Failure Scenario:

The test fails if the book management list is empty or if any step of the process encounters an issue.

Thread 15: Test Open Manage Books List After Error in Book Addition ()

The purpose of this unit test is to ensure that the system handles errors appropriately when an administrator attempts to add a new book with invalid data, and subsequently, the book management list remains unchanged. The test checks whether the system displays an error message and confirms that the book management list is not updated after encountering an error in book addition.

Success Scenario:

The test passes if the system displays an error message, and the book management list remains unchanged after attempting to add a book with an error.

Failure Scenario:

The test fails if the book management list is updated or if any step of the process encounters an issue.

Thread 16: Test Opening New Bill and Generating a Bill Successfully ()

The purpose of this unit test is to ensure that the system allows an administrator to open a new bill, add a book, generate the bill successfully, and displays the expected success message.

Success Scenario:

The test passes if the success message is displayed after generating the bill.

Failure Scenario:

The test fails if the success message is not visible, or if any step of the process encounters an issue.

Thread 17: Test Open New Bill and Generating Empty Bill Error ()

The purpose of this unit test is to ensure that the system appropriately handles the case where an administrator tries to generate a bill without adding any items, displaying the expected error message.

Success Scenario:

The test passes if the error message is displayed after attempting to generate an empty bill.

Failure Scenario:

The test fails if the error message is not visible, or if any step of the process encounters an issue.

Thread 18: Test Open Employee List Editing Employee Saving Changes Successfully ()

The purpose of this unit test is to ensure that the system allows an administrator to register a new employee, open the employee management list, edit the employee details, save the changes successfully, and displays the expected success message.

Success Scenario:

The test passes if the success message is displayed after editing and saving employee details.

Failure Scenario:

The test fails if the success message is not visible, or if any step of the process encounters an issue.

Thread 19: Test Open Employee List Attempting Empty Edit, and Checking for Error ()

The purpose of this unit test is to ensure that the system appropriately handles the case where an administrator tries to edit an employee's details with empty credentials, displaying the expected error message.

Success Scenario:

The test passes if the error message is displayed after attempting an empty edit.

Failure Scenario:

The test fails if the error message is not visible, or if any step of the process encounters an issue.

Thread 20: Test Open Employee List Deleting Employee, and Checking for Updated State ()

The purpose of this unit test is to ensure that the system allows an administrator to register a new employee, open the employee management list, delete the registered employee, and checks whether the employee list is updated correctly.

Success Scenario:

The test passes if the employee list is updated and empty after the deletion.

Failure Scenario:

The test fails if the employee list is not updated correctly, or if any step of the process encounters an issue.

Thread 21: Test Open Edit Books List Success Updated After Book Edit()

The purpose of this unit test is to simulate the scenario where an administrator adds a new book, edits its title, and checks whether the system displays a success message.

Success Scenario:

The test passes if the success message is displayed after the administrator successfully edits the book title.

Failure Scenario:

The test fails if the success message is not displayed or if any step of the process encounters an issue.

Thread 22: Test Open Delete Book Checking List Updated After Book Edit ()

The purpose of this unit test is to simulate the scenario where an administrator adds a new book, deletes it, and checks whether the table of books is updated after the deletion.

Success Scenario:

The test passes if the books table is empty after the administrator successfully deletes the added book.

Failure Scenario:

The test fails if the books table is not empty or if any step of the process encounters an issue.

Manager Panel Test (these threads happen when you log in with manager Credentials)

Thread 23: Test Add New Book Success ()

The purpose of this unit test is to ensure that the system allows a manager to successfully add a new book. The test checks whether the system correctly handles the input of book details, submits the information, and displays a success message.

Success Scenario:

The test passes if the system successfully adds the new book, and the confirmation message is visible.

Failure Scenario:

The test fails if the confirmation message is not visible or if any step of the process encounters an issue.

Thread 24: Test Add New Book Empty Credentials ()

The purpose of this unit test is to ensure that the system handles the scenario where a manager attempts to add a new book with empty credentials or fields. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects empty credentials and displays the expected error message.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 25: Test Add New Book Exception Thrown ()

The purpose of this unit test is to ensure that the system appropriately handles exceptions thrown during the attempt to add a new book by a manager. The test checks whether the system displays an appropriate error message.

Success Scenario:

The test passes if the system correctly detects the exception (in this case, due to a non-numeric purchased price) and displays the expected error message with an "OK" button.

Failure Scenario:

The test fails if the error message is not visible or if any step of the process encounters an issue.

Thread 26: Test Open Manage Books List Success Updated After Book Addition ()

The purpose of this unit test is to ensure that the system updates the book management list successfully after a new book is added. The test checks whether the system correctly displays the newly added book in the book management list.

Success Scenario:

The test passes if the book management list is successfully updated, and it is not empty after adding a new book.

Failure Scenario:

The test fails if the book management list is empty or if any step of the process encounters an issue.

Thread 27: Test Open Manage Books List After Error in Book Addition ()

The purpose of this unit test is to ensure that the system handles errors appropriately when an manager attempts to add a new book with invalid data, and subsequently, the book management list remains unchanged. The test checks whether the system displays an error message and confirms that the book management list is not updated after encountering an error in book addition.

Success Scenario:

The test passes if the system displays an error message, and the book management list remains unchanged after attempting to add a book with an error.

Failure Scenario:

The test fails if the book management list is updated or if any step of the process encounters an issue.

Thread 28: Test Open Librarian Stats View List ()

The purpose of this unit test is to simulate an attempt by a manager to open the statistics view focused on librarians and check whether the statistics table is populated.

Success Scenario:

The test passes if the statistics table for librarians is not empty after the administrator successfully opens the statistics view.

Failure Scenario:

The test fails if the statistics table is empty or if any step of the process encounters an issue.

Thread 29: Test Open Edit Books List Success Updated After Book Edit()

The purpose of this unit test is to simulate the scenario where an menager adds a new book, edits its title, and checks whether the system displays a success message.

Success Scenario:

The test passes if the success message is displayed after the administrator successfully edits the book title.

Failure Scenario:

The test fails if the success message is not displayed or if any step of the process encounters an issue.

Thread 30: Test Open Delete Book Checking List Updated After Book Edit ()

The purpose of this unit test is to simulate the scenario where an menager adds a new book, deletes it, and checks whether the table of books is updated after the deletion.

Success Scenario:

The test passes if the books table is empty after the administrator successfully deletes the added book.

Failure Scenario:

The test fails if the books table is not empty or if any step of the process encounters an issue.

Librarian Panel Test (these threads happen when you log in with librarian Credentials)

Thread 31: Test Opening New Bill and Generating a Bill Successfully ()

The purpose of this unit test is to ensure that the system allows a librarian to open a new bill, add a book, generate the bill successfully, and displays the expected success message.

Success Scenario:

The test passes if the success message is displayed after generating the bill.

Failure Scenario:

The test fails if the success message is not visible, or if any step of the process encounters an issue.

Thread 32: Test Open New Bill and Generating Empty Bill Error ()

The purpose of this unit test is to ensure that the system appropriately handles the case where a librarian tries to generate a bill without adding any items, displaying the expected error message.

Success Scenario:

The test passes if the error message is displayed after attempting to generate an empty bill.

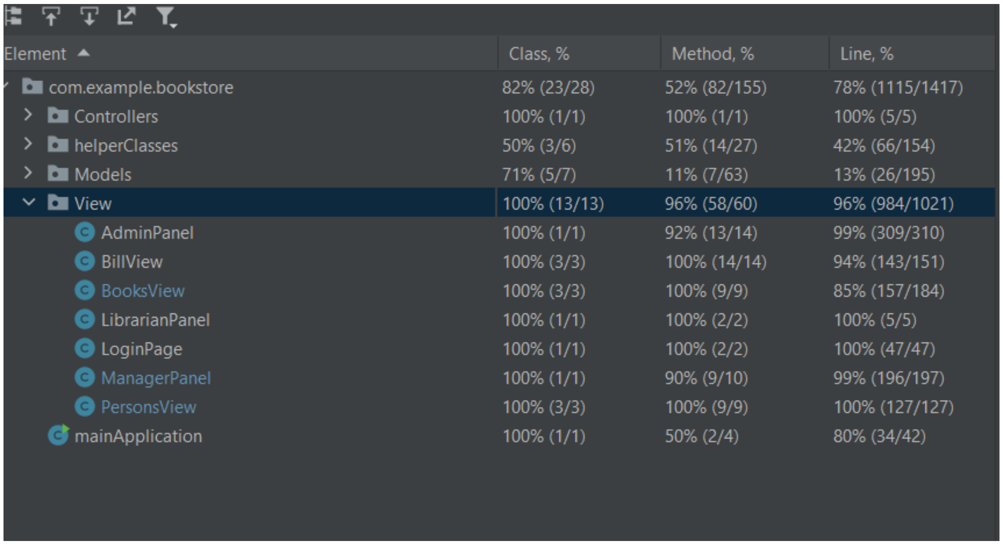
Failure Scenario:

The test fails if the error message is not visible, or if any step of the process encounters an issue.

Test Coverage Metric

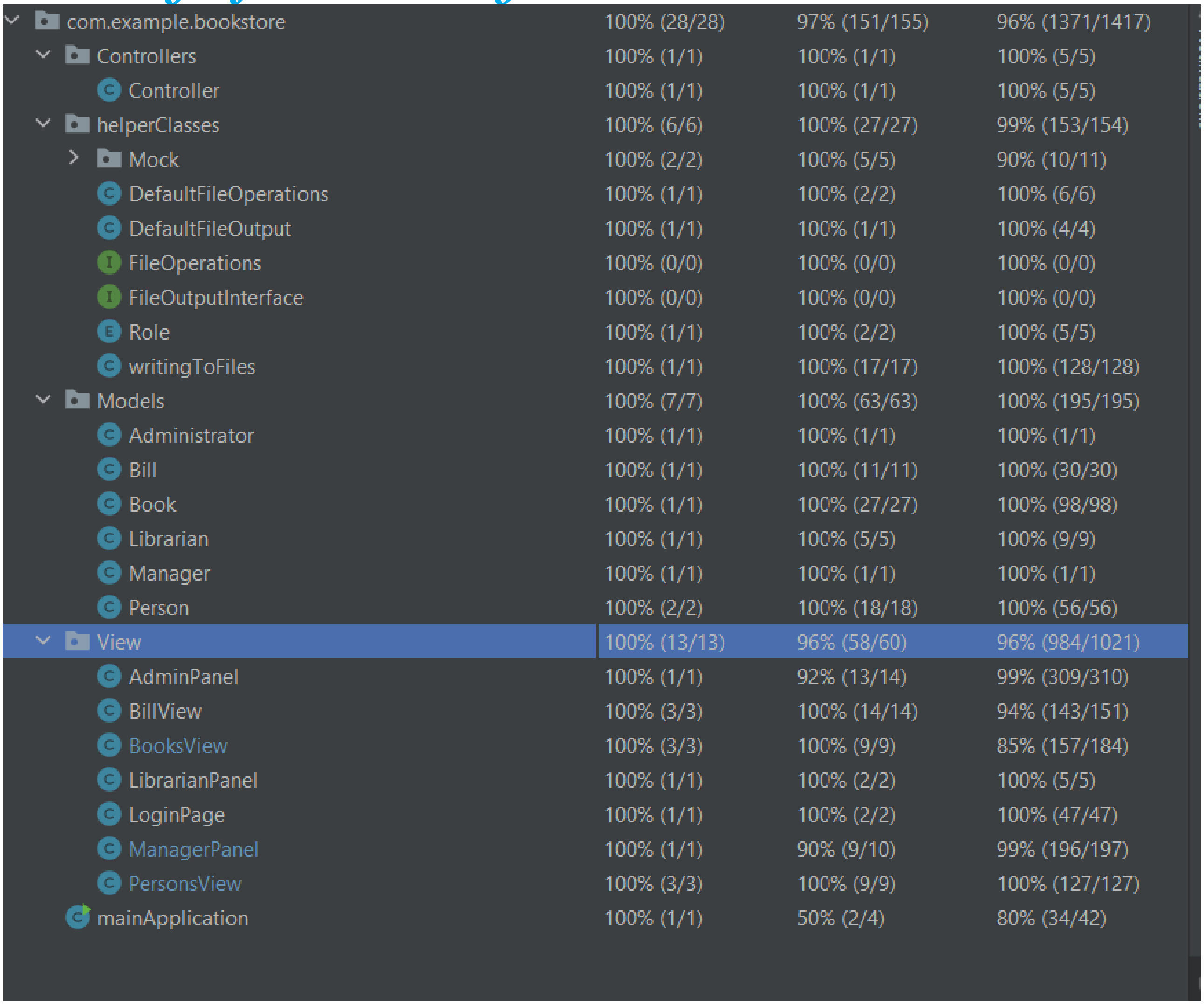
I have chosen to take line coverage because achieving high line coverage is often associated with a comprehensive testing strategy, which can contribute to overall code quality. Thoroughly tested code is more likely to be robust, reliable, and less prone to defects. It ensures that every line of code has been touched at least once, making it easier to identify areas that have not been tested. My total line coverage percentage is 96%, this is because in system testing, I have done 32 threads (the most important ones) which have made the percentage drop.

Total Coverage of all the System Testing



Element	Class, %	Method, %	Line, %
com.example.bookstore	82% (23/28)	52% (82/155)	78% (1115/1417)
Controllers	100% (1/1)	100% (1/1)	100% (5/5)
helperClasses	50% (3/6)	51% (14/27)	42% (66/154)
Models	71% (5/7)	11% (7/63)	13% (26/195)
View	100% (13/13)	96% (58/60)	96% (984/1021)
AdminPanel	100% (1/1)	92% (13/14)	99% (309/310)
BillView	100% (3/3)	100% (14/14)	94% (143/151)
BooksView	100% (3/3)	100% (9/9)	85% (157/184)
LibrarianPanel	100% (1/1)	100% (2/2)	100% (5/5)
LoginPage	100% (1/1)	100% (2/2)	100% (47/47)
ManagerPanel	100% (1/1)	90% (9/10)	99% (196/197)
PersonsView	100% (3/3)	100% (9/9)	100% (127/127)
mainApplication	100% (1/1)	50% (2/4)	80% (34/42)

Total Coverage of all the Testing



com.example.bookstore	100% (28/28)	97% (151/155)	96% (1371/1417)
Controllers	100% (1/1)	100% (1/1)	100% (5/5)
Controller	100% (1/1)	100% (1/1)	100% (5/5)
helperClasses	100% (6/6)	100% (27/27)	99% (153/154)
Mock	100% (2/2)	100% (5/5)	90% (10/11)
DefaultFileOperations	100% (1/1)	100% (2/2)	100% (6/6)
DefaultFileOutput	100% (1/1)	100% (1/1)	100% (4/4)
FileOperations	100% (0/0)	100% (0/0)	100% (0/0)
FileOutputInterface	100% (0/0)	100% (0/0)	100% (0/0)
Role	100% (1/1)	100% (2/2)	100% (5/5)
writingToFiles	100% (1/1)	100% (17/17)	100% (128/128)
Models	100% (7/7)	100% (63/63)	100% (195/195)
Administrator	100% (1/1)	100% (1/1)	100% (1/1)
Bill	100% (1/1)	100% (11/11)	100% (30/30)
Book	100% (1/1)	100% (27/27)	100% (98/98)
Librarian	100% (1/1)	100% (5/5)	100% (9/9)
Manager	100% (1/1)	100% (1/1)	100% (1/1)
Person	100% (2/2)	100% (18/18)	100% (56/56)
View	100% (13/13)	96% (58/60)	96% (984/1021)
AdminPanel	100% (1/1)	92% (13/14)	99% (309/310)
BillView	100% (3/3)	100% (14/14)	94% (143/151)
BooksView	100% (3/3)	100% (9/9)	85% (157/184)
LibrarianPanel	100% (1/1)	100% (2/2)	100% (5/5)
LoginPage	100% (1/1)	100% (2/2)	100% (47/47)
ManagerPanel	100% (1/1)	90% (9/10)	99% (196/197)
PersonsView	100% (3/3)	100% (9/9)	100% (127/127)
mainApplication	100% (1/1)	50% (2/4)	80% (34/42)

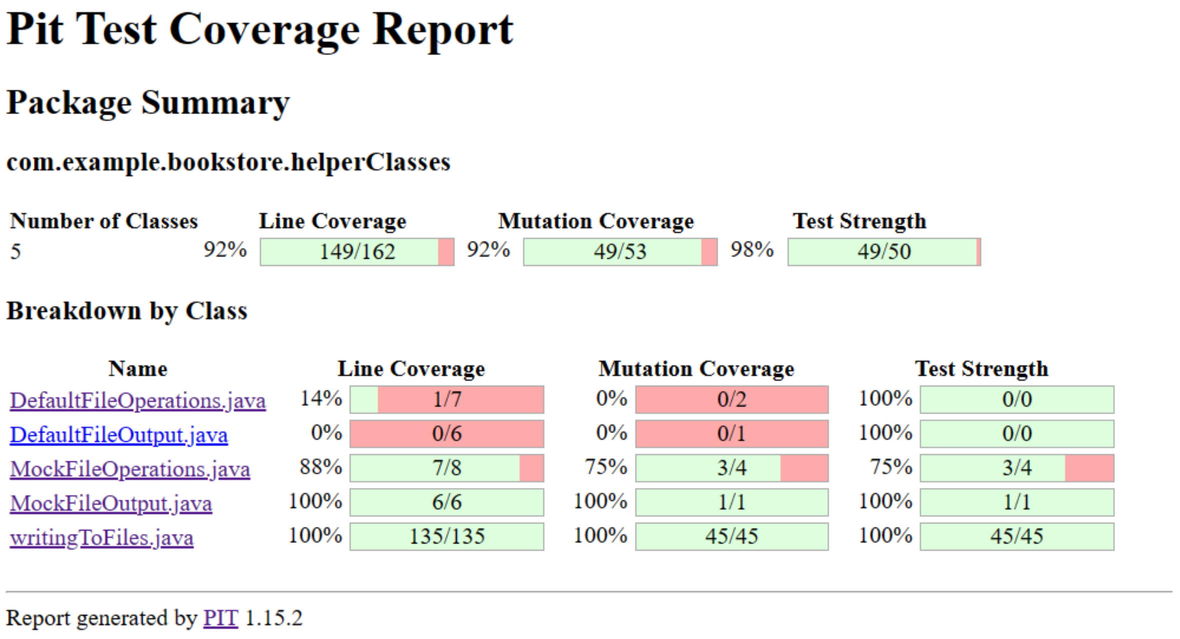
Mutation Testing

Model classes

PersonTest: Test strength 97%
LibrarianTest: Test strength 100%
BookTest: Test strength 95%
LibrarianTest: Test strength 74%

Helper classes

Test Writing to files Unit ()



Pit Test Coverage Report

Package Summary

com.example.bookstore.helperClasses

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
5	92%	149/162	92%	49/53	98%	49/50

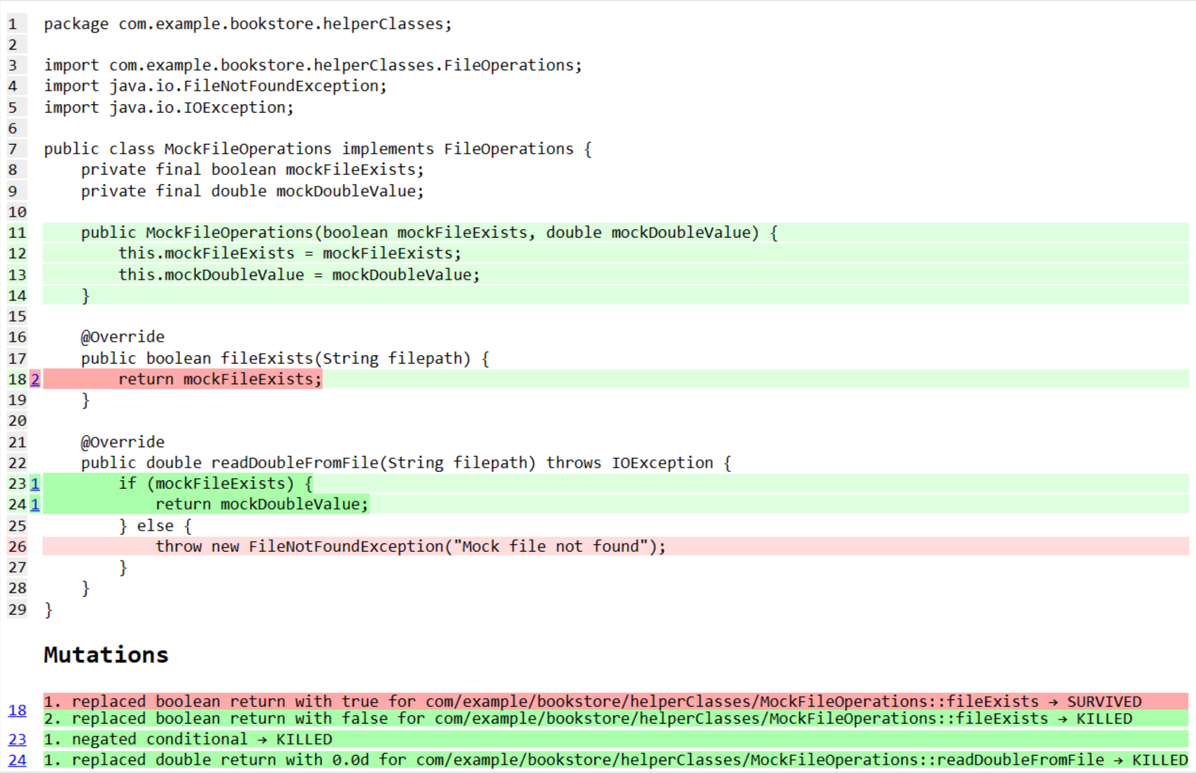
Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
DefaultFileOperations.java	14%	1/7	0%	0/2	100%	0/0
DefaultFileOutput.java	0%	0/6	0%	0/1	100%	0/0
MockFileOperations.java	88%	7/8	75%	3/4	75%	3/4
MockFileOutput.java	100%	6/6	100%	1/1	100%	1/1
writingToFiles.java	100%	135/135	100%	45/45	100%	45/45

Report generated by PIT 1.15.2

MockFileOperations:
The test for mock File Exists for true value is supposed to survive since the class is a mock and what we pass to it is going to be considered as true. When true it just returns the mock double Value.

MockFileOperations.java



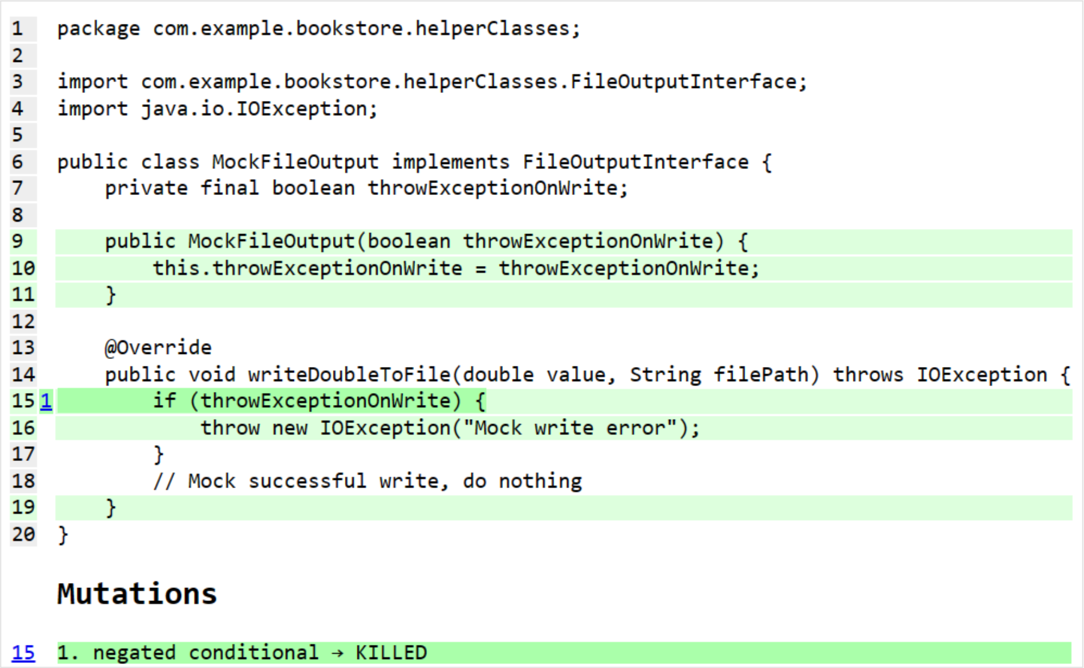
```
1     package com.example.bookstore.helperClasses;
2
3     import com.example.bookstore.helperClasses.FileOperations;
4     import java.io.FileNotFoundException;
5     import java.io.IOException;
6
7     public class MockFileOperations implements FileOperations {
8         private final boolean mockFileExists;
9         private final double mockDoubleValue;
10
11        public MockFileOperations(boolean mockFileExists, double mockDoubleValue) {
12            this.mockFileExists = mockFileExists;
13            this.mockDoubleValue = mockDoubleValue;
14        }
15
16        @Override
17        public boolean fileExists(String filepath) {
18 2          return mockFileExists;
19        }
20
21        @Override
22        public double readDoubleFromFile(String filepath) throws IOException {
23 1          if (mockFileExists) {
24 1              return mockDoubleValue;
25            } else {
26                throw new FileNotFoundException("Mock file not found");
27            }
28        }
29    }
```

Mutations

```
18  1. replaced boolean return with true for com/example/bookstore/helperClasses/MockFileOperations::fileExists → SURVIVED
    2. replaced boolean return with false for com/example/bookstore/helperClasses/MockFileOperations::fileExists → KILLED
23  1. negated conditional → KILLED
24  1. replaced double return with 0.0d for com/example/bookstore/helperClasses/MockFileOperations::readDoubleFromFile → KILLED
```

MockFileOutput:
All tests were killed.

MockFileOutput.java



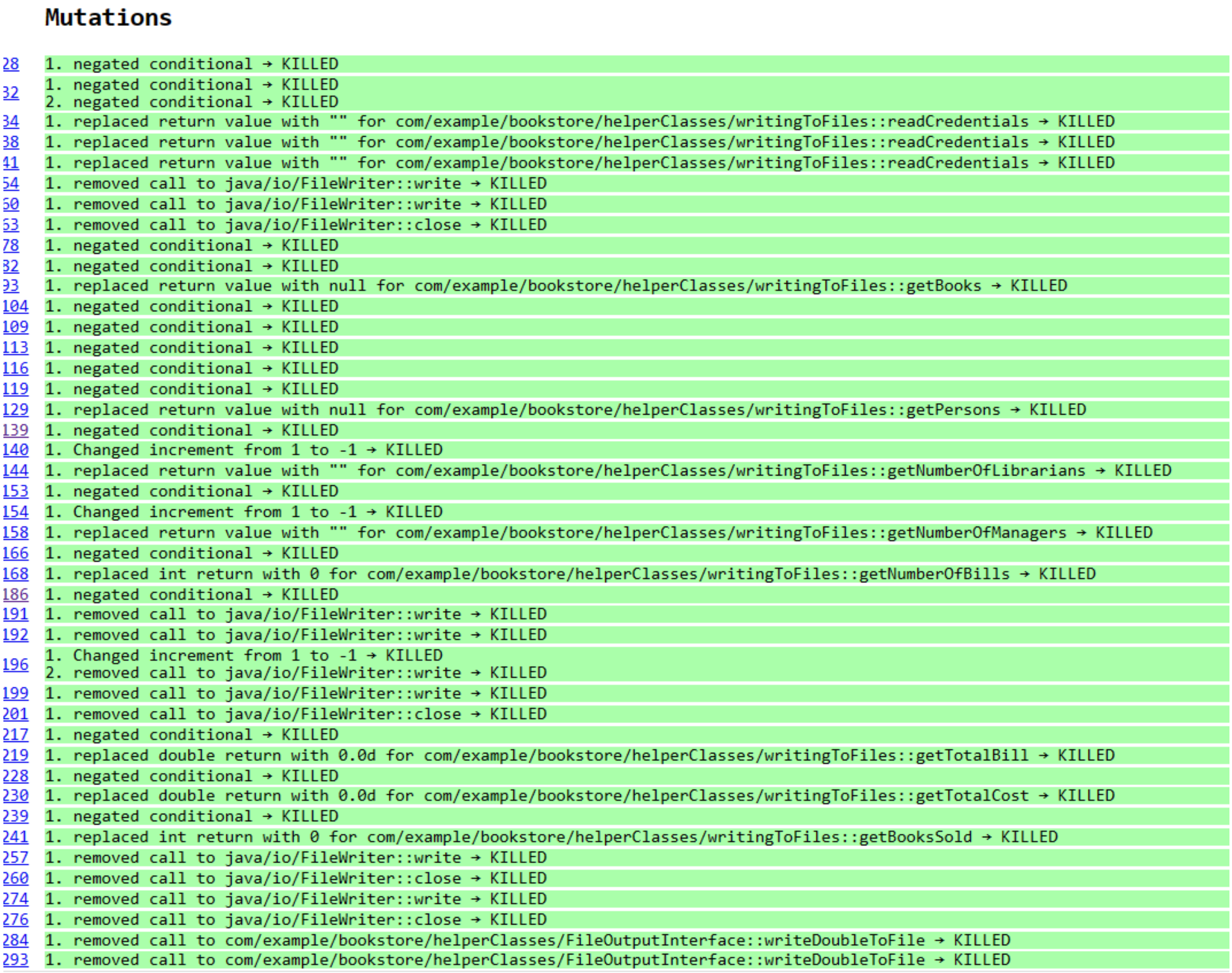
```
1     package com.example.bookstore.helperClasses;
2
3     import com.example.bookstore.helperClasses.FileOutputInterface;
4     import java.io.IOException;
5
6     public class MockFileOutput implements FileOutputInterface {
7         private final boolean throwExceptionOnWrite;
8
9         public MockFileOutput(boolean throwExceptionOnWrite) {
10            this.throwExceptionOnWrite = throwExceptionOnWrite;
11        }
12
13        @Override
14        public void writeDoubleToFile(double value, String filePath) throws IOException {
15 1          if (throwExceptionOnWrite) {
16                throw new IOException("Mock write error");
17            }
18            // Mock successful write, do nothing
19        }
20    }
```

Mutations

```
15  1. negated conditional → KILLED
```

WrittingToFiles:

All tests were killed.

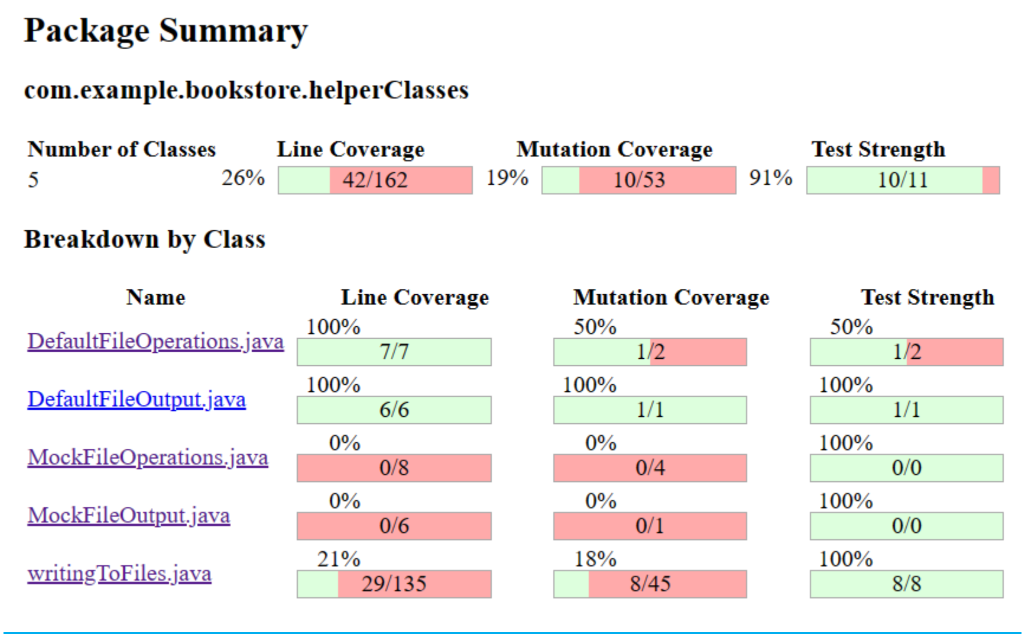


```
Mutations

28  1. negated conditional → KILLED
32  1. negated conditional → KILLED
    2. negated conditional → KILLED
34  1. replaced return value with "" for com/example/bookstore/helperClasses/writingToFiles::readCredentials → KILLED
38  1. replaced return value with "" for com/example/bookstore/helperClasses/writingToFiles::readCredentials → KILLED
41  1. replaced return value with "" for com/example/bookstore/helperClasses/writingToFiles::readCredentials → KILLED
54  1. removed call to java/io/FileWriter::write → KILLED
60  1. removed call to java/io/FileWriter::write → KILLED
63  1. removed call to java/io/FileWriter::close → KILLED
78  1. negated conditional → KILLED
82  1. negated conditional → KILLED
93  1. replaced return value with null for com/example/bookstore/helperClasses/writingToFiles::getBooks → KILLED
104 1. negated conditional → KILLED
109 1. negated conditional → KILLED
113 1. negated conditional → KILLED
116 1. negated conditional → KILLED
119 1. negated conditional → KILLED
129 1. replaced return value with null for com/example/bookstore/helperClasses/writingToFiles::getPersons → KILLED
139 1. negated conditional → KILLED
140 1. Changed increment from 1 to -1 → KILLED
144 1. replaced return value with "" for com/example/bookstore/helperClasses/writingToFiles::getNumberOfLibrarians → KILLED
153 1. negated conditional → KILLED
154 1. Changed increment from 1 to -1 → KILLED
158 1. replaced return value with "" for com/example/bookstore/helperClasses/writingToFiles::getNumberOfManagers → KILLED
166 1. negated conditional → KILLED
168 1. replaced int return with 0 for com/example/bookstore/helperClasses/writingToFiles::getNumberOfBills → KILLED
186 1. negated conditional → KILLED
191 1. removed call to java/io/FileWriter::write → KILLED
192 1. removed call to java/io/FileWriter::write → KILLED
196 1. Changed increment from 1 to -1 → KILLED
    2. removed call to java/io/FileWriter::write → KILLED
199 1. removed call to java/io/FileWriter::write → KILLED
201 1. removed call to java/io/FileWriter::close → KILLED
217 1. negated conditional → KILLED
219 1. replaced double return with 0.0d for com/example/bookstore/helperClasses/writingToFiles::getTotalBill → KILLED
228 1. negated conditional → KILLED
230 1. replaced double return with 0.0d for com/example/bookstore/helperClasses/writingToFiles::getTotalCost → KILLED
239 1. negated conditional → KILLED
241 1. replaced int return with 0 for com/example/bookstore/helperClasses/writingToFiles::getBooksSold → KILLED
257 1. removed call to java/io/FileWriter::write → KILLED
260 1. removed call to java/io/FileWriter::close → KILLED
274 1. removed call to java/io/FileWriter::write → KILLED
276 1. removed call to java/io/FileWriter::close → KILLED
284 1. removed call to com/example/bookstore/helperClasses/FileOutputInterface::writeDoubleToFile → KILLED
293 1. removed call to com/example/bookstore/helperClasses/FileOutputInterface::writeDoubleToFile → KILLED
```

!! We don’t check the classes Default File Operations and Output Because they aren’t used in unit testing only mocks are.

Test Writing to files Integration ()



Package Summary

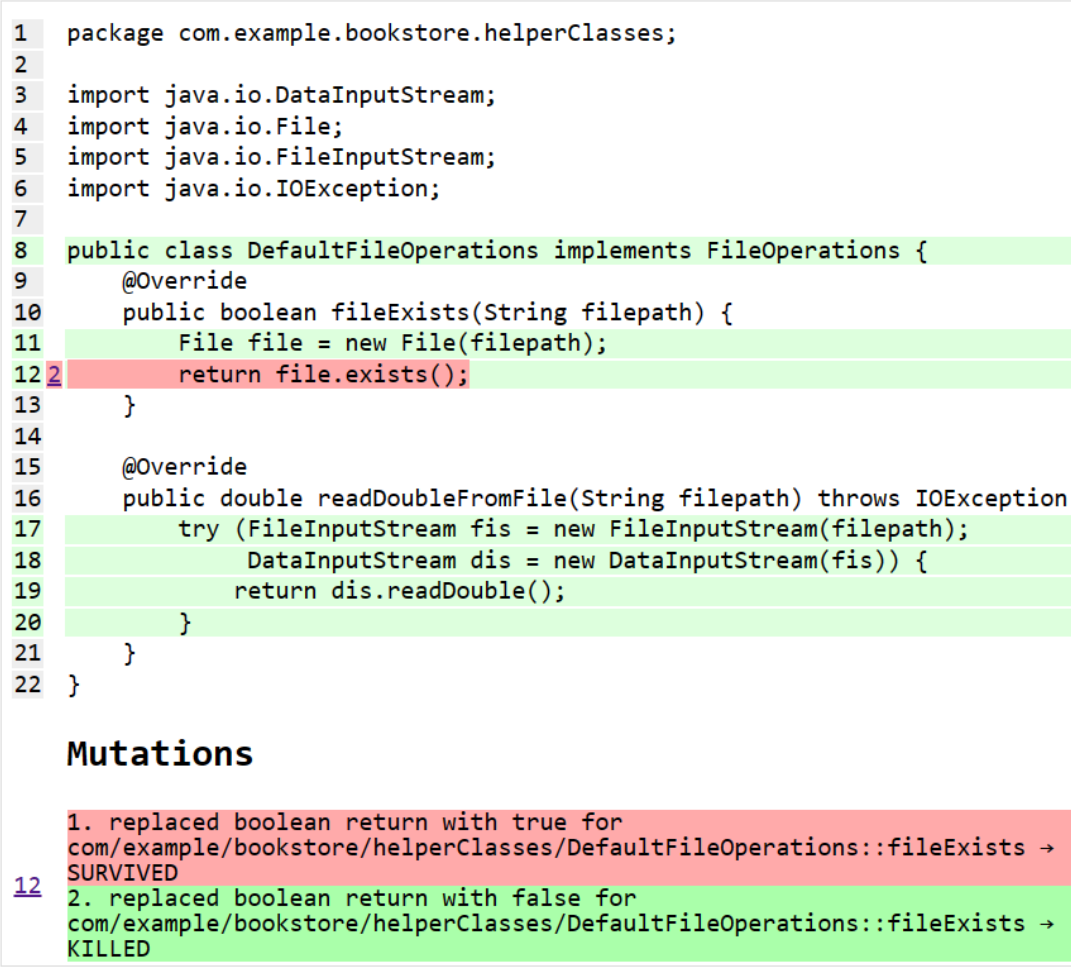
com.example.bookstore.helperClasses

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
5	26%	42/162	19%	10/53	91%	10/11

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
DefaultFileOperations.java	100%	7/7	50%	1/2	50%	1/2
DefaultFileOutput.java	100%	6/6	100%	1/1	100%	1/1
MockFileOperations.java	0%	0/8	0%	0/4	100%	0/0
MockFileOutput.java	0%	0/6	0%	0/1	100%	0/0
writingToFiles.java	21%	29/135	18%	8/45	100%	8/8

DefaultFileOperations:
The test for true value is supposed to survive



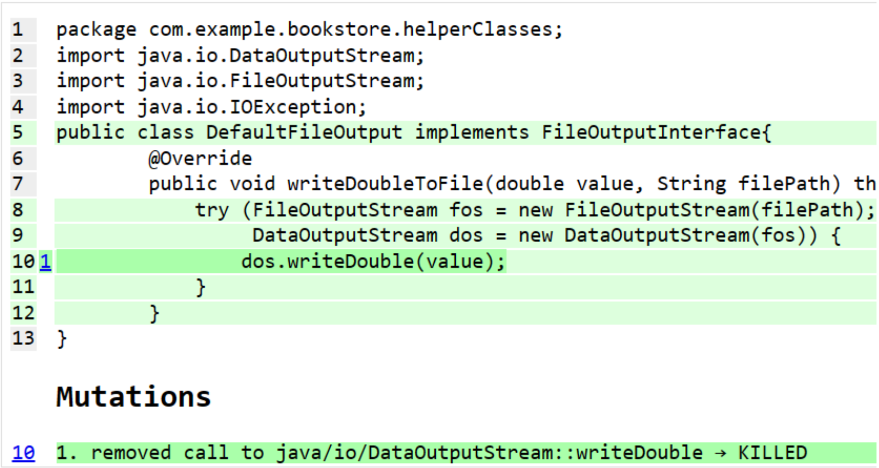
```
1    package com.example.bookstore.helperClasses;
2
3    import java.io.DataInputStream;
4    import java.io.File;
5    import java.io.FileInputStream;
6    import java.io.IOException;
7
8    public class DefaultFileOperations implements FileOperations {
9        @Override
10       public boolean fileExists(String filepath) {
11           File file = new File(filepath);
12 2         return file.exists();
13       }
14
15       @Override
16       public double readDoubleFromFile(String filepath) throws IOException
17           try (FileInputStream fis = new FileInputStream(filepath);
18                DataInputStream dis = new DataInputStream(fis)) {
19               return dis.readDouble();
20           }
21       }
22   }
```

Mutations

```
12   1. replaced boolean return with true for
     com/example/bookstore/helperClasses/DefaultFileOperations::fileExists →
     SURVIVED
     2. replaced boolean return with false for
     com/example/bookstore/helperClasses/DefaultFileOperations::fileExists →
     KILLED
```

DefaultFileOutput:
All tests survive.

DefaultFileOutput.java

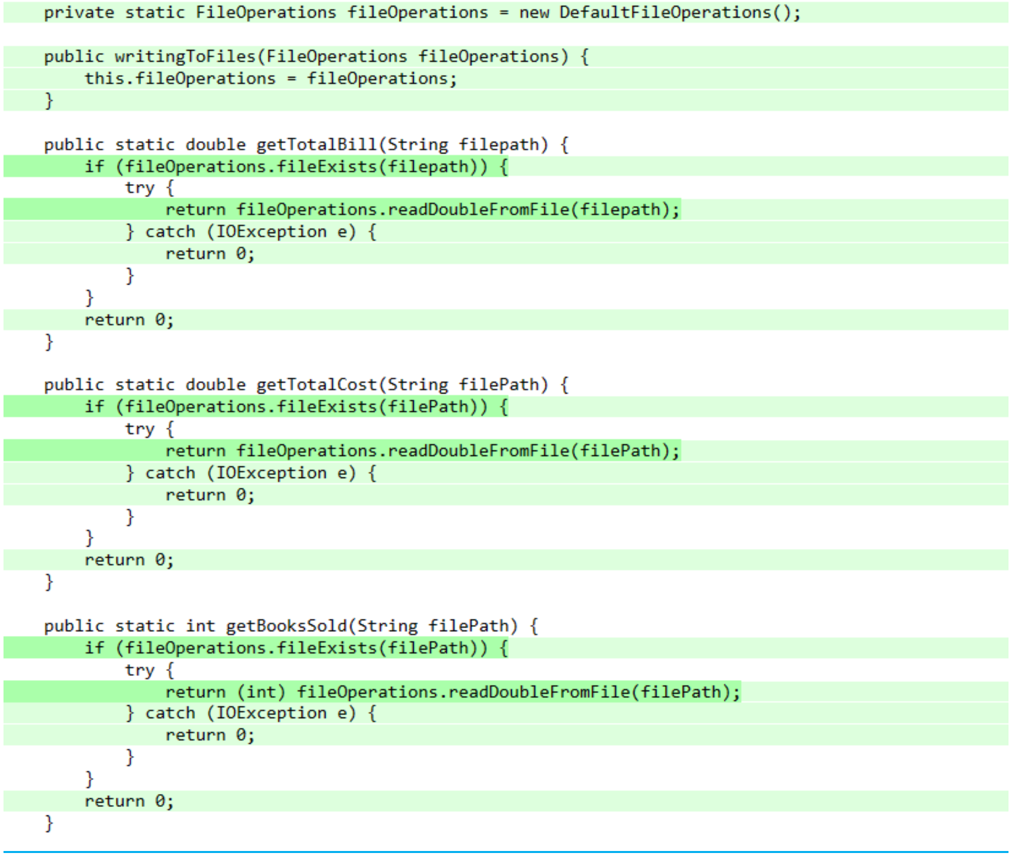


```
1   package com.example.bookstore.helperClasses;
2   import java.io.DataOutputStream;
3   import java.io.FileOutputStream;
4   import java.io.IOException;
5   public class DefaultFileOutput implements FileOutputInterface{
6           @Override
7           public void writeDoubleToFile(double value, String filePath) th
8               try (FileOutputStream fos = new FileOutputStream(filePath);
9                    DataOutputStream dos = new DataOutputStream(fos)) {
10 1               dos.writeDouble(value);
11              }
12          }
13  }
```

Mutations

```
10  1. removed call to java/io/DataOutputStream::writeDouble → KILLED
```

Writtetofiles:
All tests for methods we are integration testing survive.



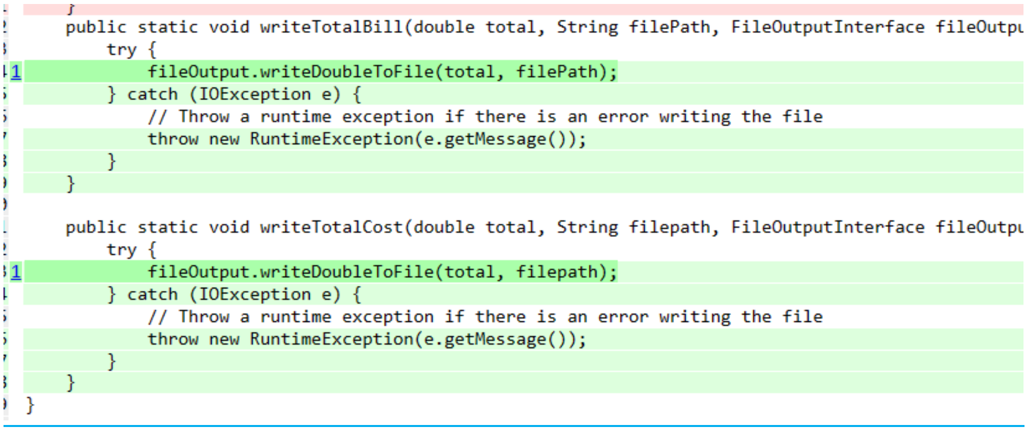
```
    private static FileOperations fileOperations = new DefaultFileOperations();

    public writingToFiles(FileOperations fileOperations) {
        this.fileOperations = fileOperations;
    }

    public static double getTotalBill(String filepath) {
        if (fileOperations.fileExists(filepath)) {
            try {
                return fileOperations.readDoubleFromFile(filepath);
            } catch (IOException e) {
                return 0;
            }
        }
        return 0;
    }

    public static double getTotalCost(String filePath) {
        if (fileOperations.fileExists(filePath)) {
            try {
                return fileOperations.readDoubleFromFile(filePath);
            } catch (IOException e) {
                return 0;
            }
        }
        return 0;
    }

    public static int getBooksSold(String filePath) {
        if (fileOperations.fileExists(filePath)) {
            try {
                return (int) fileOperations.readDoubleFromFile(filePath);
            } catch (IOException e) {
                return 0;
            }
        }
        return 0;
    }
```



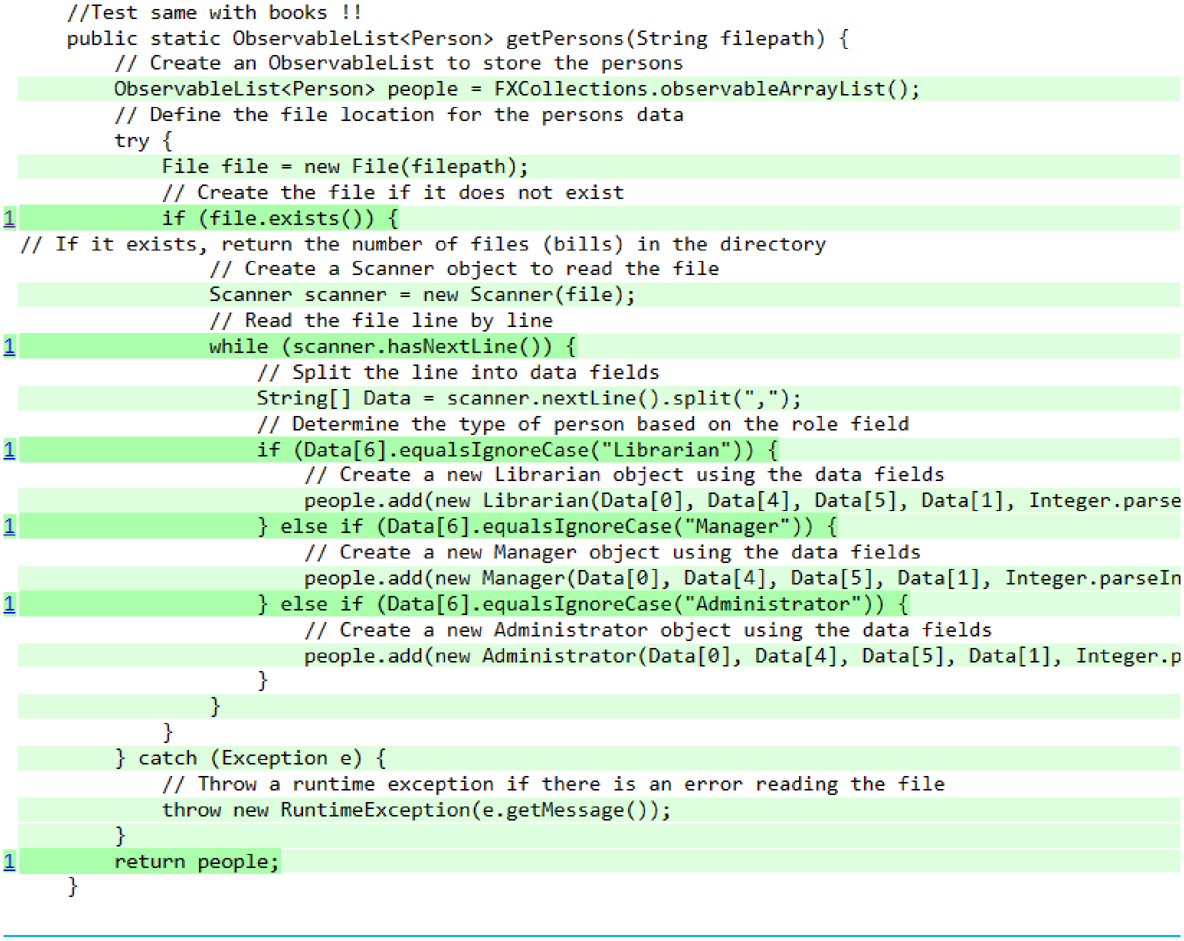
```
    }
    public static void writeTotalBill(double total, String filePath, FileOutputInterface fileOutpu
        try {
1           fileOutput.writeDoubleToFile(total, filePath);
        } catch (IOException e) {
            // Throw a runtime exception if there is an error writing the file
            throw new RuntimeException(e.getMessage());
        }
    }

    public static void writeTotalCost(double total, String filepath, FileOutputInterface fileOutpu
        try {
1           fileOutput.writeDoubleToFile(total, filepath);
        } catch (IOException e) {
            // Throw a runtime exception if there is an error writing the file
            throw new RuntimeException(e.getMessage());
        }
    }
}
```

!! We don’t check the classes Mock File Operations and Output Because they aren’t used in integration testing.

Test Basis Path for Get Persons()

Writtetofiles:
All tests for method get persons survive.



```
    //Test same with books !!
    public static ObservableList<Person> getPersons(String filepath) {
        // Create an ObservableList to store the persons
        ObservableList<Person> people = FXCollections.observableArrayList();
        // Define the file location for the persons data
        try {
            File file = new File(filepath);
            // Create the file if it does not exist
1           if (file.exists()) {
// If it exists, return the number of files (bills) in the directory
                // Create a Scanner object to read the file
                Scanner scanner = new Scanner(file);
                // Read the file line by line
1               while (scanner.hasNextLine()) {
                    // Split the line into data fields
                    String[] Data = scanner.nextLine().split(",");
                    // Determine the type of person based on the role field
1                   if (Data[6].equalsIgnoreCase("Librarian")) {
                        // Create a new Librarian object using the data fields
                        people.add(new Librarian(Data[0], Data[4], Data[5], Data[1], Integer.parse
1                   } else if (Data[6].equalsIgnoreCase("Manager")) {
                        // Create a new Manager object using the data fields
                        people.add(new Manager(Data[0], Data[4], Data[5], Data[1], Integer.parseIn
1                   } else if (Data[6].equalsIgnoreCase("Administrator")) {
                        // Create a new Administrator object using the data fields
                        people.add(new Administrator(Data[0], Data[4], Data[5], Data[1], Integer.p
                    }
                }
            }
        } catch (Exception e) {
            // Throw a runtime exception if there is an error reading the file
            throw new RuntimeException(e.getMessage());
        }
1       return people;
    }
```

!! We don’t check other classes and methods because this test only covers get persons.

Work done by every member:

Qodana is done by Tea Malasi.

Classes	Kevin Llaca	Tea Malasi
(Helper classes) Basis Path Testing for Get Persons	+	
(Helper classes) Testing Writing to files Integration	6 methods	7 methods
(Helper classes) Testing Writing to files Unit Testing	25 methods	34 methods
(Models) Bill Test	+	
(Models) Book Test	16 methods	41 methods
(Models) Librarian Test	+	
(Models) Persons Test	30 methods	2 methods
(View) Admin Panel Test	8 methods	9 methods
(View) Librarian Panel Test		+
(View) Manager Panel Test	6methods	2 methods
Main Application Test		+

Pit is done by Tea Malasi.
Report is done by Kevin Llaca and Tea Malasi.