Software Testing & Quality Assurance Project Report

Project: Electronics Store
Course: Software Testing & Quality Assurance

Team Members:

Enea Cane

Jurgen Hila

Orgest Baçova

Sidrit Zela

Xhois Cano

1. Introduction

This report documents the testing activities performed on the **Electronics Store** project, a Java-based application. The testing process followed the guidelines outlined in the project requirements, focusing on **static testing**, **testing analysis**, and **unit, integration, and system testing**. The team worked collaboratively to ensure comprehensive coverage of the project.

---

2. Types of Testing, Documentation, and GitHub Usage

2.1 Types of Testing

- Static Testing
- Testing Analysis using Boundary Value Testing (BVT), Equivalence Class Testing and Code Coverage
- Unit, Integration, and System Testing

2.2 Documentation Style

- Each member maintained individual testing documentation.
- Each documentation for static testing can be found in the Google Sheets link: Static Testing with SonarQube.
- Every other documentation for the different types of testing can be found in the GitHub repository explained in greater detail. (Link: https://github.com/sidrit30/Electronics-Store).

2.3 GitHub Usage

- GitHub was used systematically for version control and for documenting bug fixes.
- Each member committed their work regularly.

2.4 Technologies used

- SonarQube - for static testing
- JUnit 5 - for writing the test cases
- Mockito - for creating mocks during unit testing
- TestFX - for testing JavaFX components

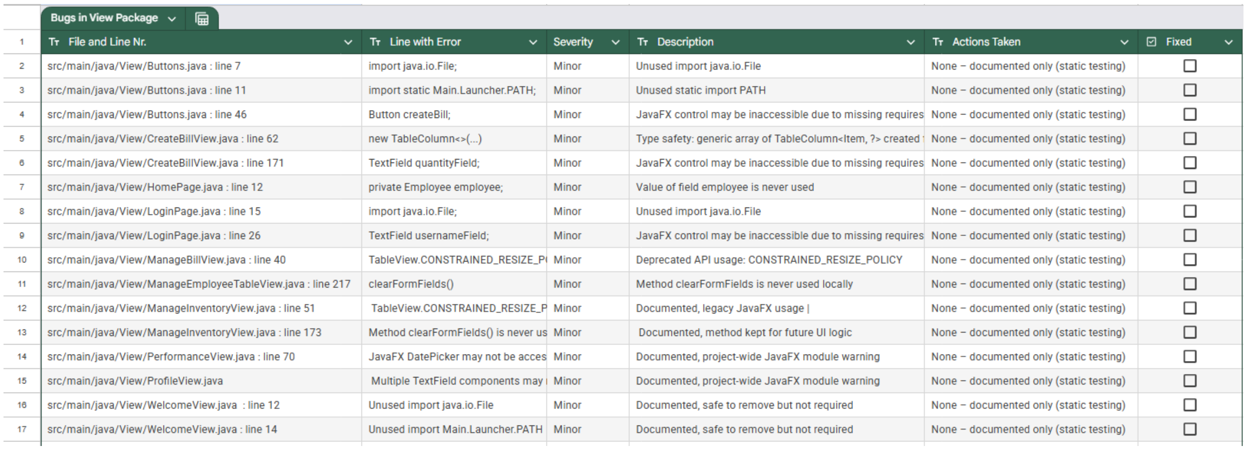
---

3. Static Testing

3.1 Static testing document

(SonarQube warnings were stored in the following spreadsheet: Static Testing with SonarQube)

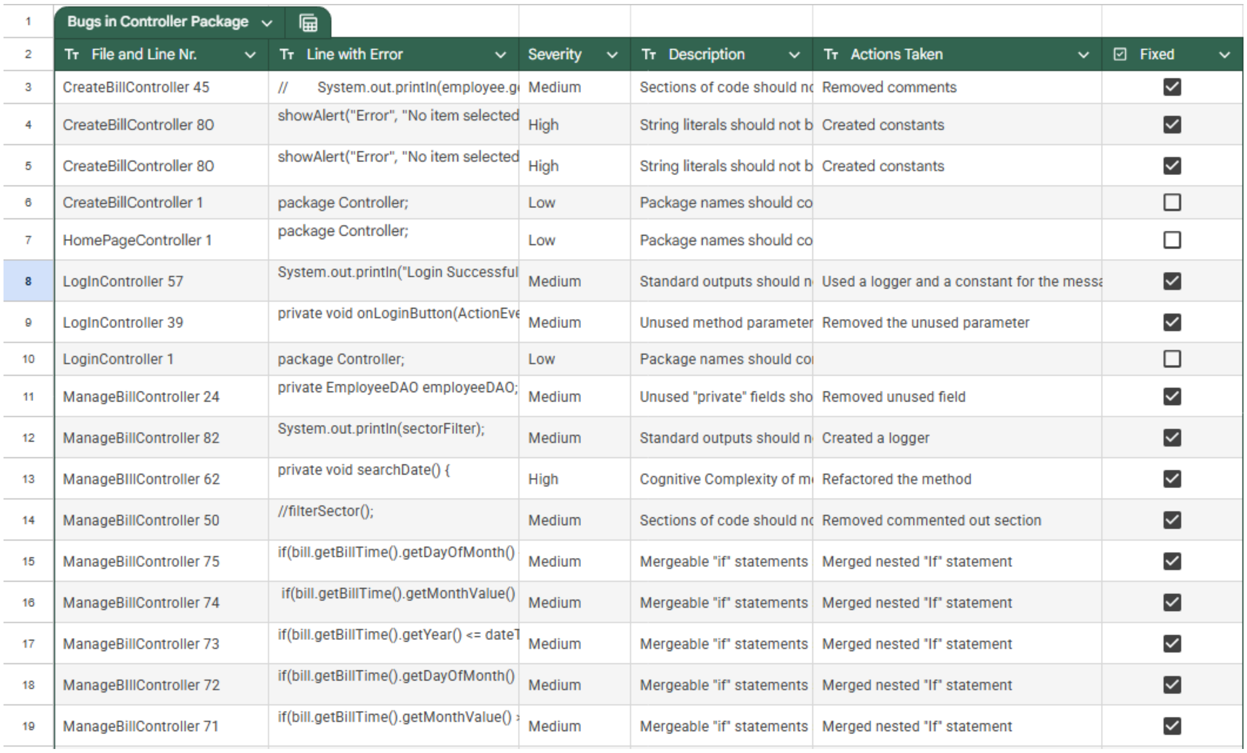
3.2 Enea Cane, Jurgen Hila - View Package



Bugs in View Package

1	File and Line Nr.	Line with Error	Severity	Description	Actions Taken	Fixed
2	src/main/java/View/Buttons.java : line 7	import java.io.File;	Minor	Unused import java.io.File	None – documented only (static testing)	☐
3	src/main/java/View/Buttons.java : line 11	import static Main.Launcher.PATH;	Minor	Unused static import PATH	None – documented only (static testing)	☐
4	src/main/java/View/Buttons.java : line 46	Button createBill;	Minor	JavaFX control may be inaccessible due to missing requires	None – documented only (static testing)	☐
5	src/main/java/View/CreateBillView.java : line 62	new TableColumn<>(...)	Minor	Type safety: generic array of TableColumn<Item, ?> created	None – documented only (static testing)	☐
6	src/main/java/View/CreateBillView.java : line 171	TextField quantityField;	Minor	JavaFX control may be inaccessible due to missing requires	None – documented only (static testing)	☐
7	src/main/java/View/HomePage.java : line 12	private Employee employee;	Minor	Value of field employee is never used	None – documented only (static testing)	☐
8	src/main/java/View/LoginPage.java : line 15	import java.io.File;	Minor	Unused import java.io.File	None – documented only (static testing)	☐
9	src/main/java/View/LoginPage.java : line 26	TextField usernameField;	Minor	JavaFX control may be inaccessible due to missing requires	None – documented only (static testing)	☐
10	src/main/java/View/ManageBillView.java : line 40	TableView.CONSTRAINED_RESIZE_P	Minor	Deprecated API usage: CONSTRAINED_RESIZE_POLICY	None – documented only (static testing)	☐
11	src/main/java/View/ManageEmployeeTableView.java : line 217	clearFormFields()	Minor	Method clearFormFields is never used locally	None – documented only (static testing)	☐
12	src/main/java/View/ManageInventoryView.java : line 51	TableView.CONSTRAINED_RESIZE_P	Minor	Documented, legacy JavaFX usage |	None – documented only (static testing)	☐
13	src/main/java/View/ManageInventoryView.java : line 173	Method clearFormFields() is never us	Minor	Documented, method kept for future UI logic	None – documented only (static testing)	☐
14	src/main/java/View/PerformanceView.java : line 70	JavaFX DatePicker may not be acces	Minor	Documented, project-wide JavaFX module warning	None – documented only (static testing)	☐
15	src/main/java/View/ProfileView.java	Multiple TextField components may	Minor	Documented, project-wide JavaFX module warning	None – documented only (static testing)	☐
16	src/main/java/View/WelcomeView.java : line 12	Unused import java.io.File	Minor	Documented, safe to remove but not required	None – documented only (static testing)	☐
17	src/main/java/View/WelcomeView.java : line 14	Unused import Main.Launcher.PATH	Minor	Documented, safe to remove but not required	None – documented only (static testing)	☐

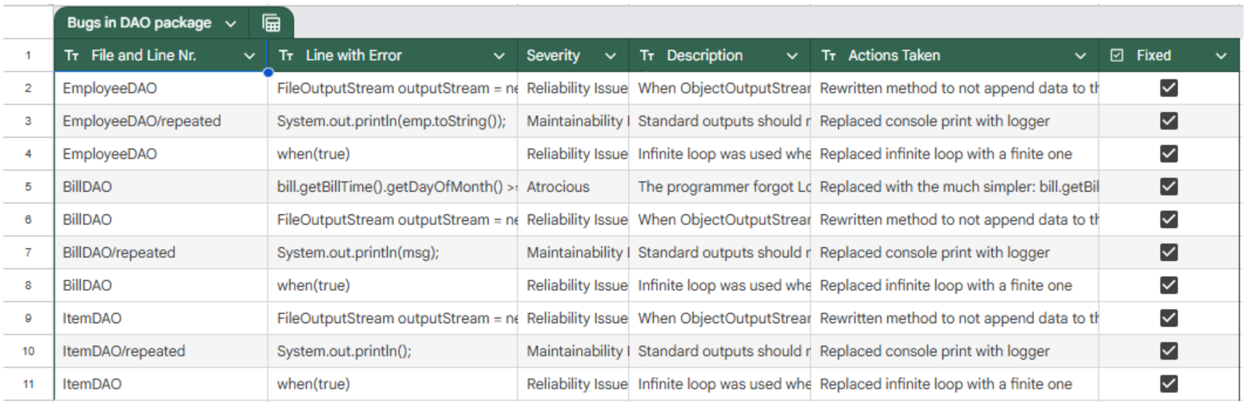
3.3 Orgest Baçova - Controller Package



Bugs in Controller Package

File and Line Nr.	Line with Error	Severity	Description	Actions Taken	Fixed
CreateBillController 45	// System.out.println(employee.g	Medium	Sections of code should nc	Removed comments	☑
CreateBillController 80	showAlert("Error", "No item selected	High	String literals should not b	Created constants	☑
CreateBillController 80	showAlert("Error", "No item selected	High	String literals should not b	Created constants	☑
CreateBillController 1	package Controller;	Low	Package names should co		☐
HomePageController 1	package Controller;	Low	Package names should co		☐
LogInController 57	System.out.println("Login Successful	Medium	Standard outputs should n	Used a logger and a constant for the messa	☑
LogInController 39	private void onLoginButton(ActionEve	Medium	Unused method parameter	Removed the unused parameter	☑
LoginController 1	package Controller;	Low	Package names should co		☐
ManageBillController 24	private EmployeeDAO employeeDAO;	Medium	Unused "private" fields sho	Removed unused field	☑
ManageBillController 82	System.out.println(sectorFilter);	Medium	Standard outputs should n	Created a logger	☑
ManageBIllController 62	private void searchDate() {	High	Cognitive Complexity of m	Refactored the method	☑
ManageBillController 50	//filterSector();	Medium	Sections of code should nc	Removed commented out section	☑
ManageBillController 75	if(bill.getBillTime().getDayOfMonth()	Medium	Mergeable "if" statements	Merged nested "If" statement	☑
ManageBillController 74	if(bill.getBillTime().getMonthValue()	Medium	Mergeable "if" statements	Merged nested "If" statement	☑
ManageBillController 73	if(bill.getBillTime().getYear() <= dateT	Medium	Mergeable "if" statements	Merged nested "If" statement	☑
ManageBIllController 72	if(bill.getBillTime().getDayOfMonth()	Medium	Mergeable "if" statements	Merged nested "If" statement	☑
ManageBillController 71	if(bill.getBillTime().getMonthValue() :	Medium	Mergeable "if" statements	Merged nested "If" statement	☑

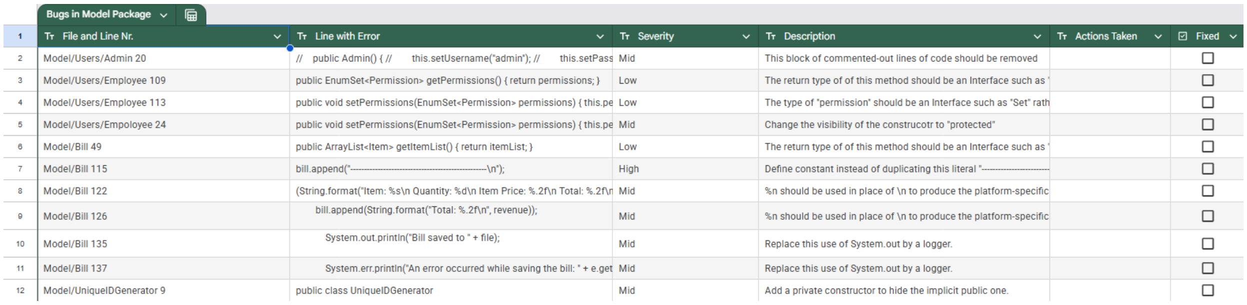
3.4 Sidrit Zela - DAO Package



Bugs in DAO package

File and Line Nr.	Line with Error	Severity	Description	Actions Taken	Fixed
EmployeeDAO	FileOutputStream outputStream = ne	Reliability Issue	When ObjectOutputStrear	Rewritten method to not append data to th	☑
EmployeeDAO/repeated	System.out.println(emp.toString());	Maintainability I	Standard outputs should r	Replaced console print with logger	☑
EmployeeDAO	when(true)	Reliability Issue	Infinite loop was used whe	Replaced infinite loop with a finite one	☑
BillDAO	bill.getBillTime().getDayOfMonth() >:	Atrocious	The programmer forgot Lc	Replaced with the much simpler: bill.getBil	☑
BillDAO	FileOutputStream outputStream = ne	Reliability Issue	When ObjectOutputStrear	Rewritten method to not append data to th	☑
BillDAO/repeated	System.out.println(msg);	Maintainability I	Standard outputs should r	Replaced console print with logger	☑
BillDAO	when(true)	Reliability Issue	Infinite loop was used whe	Replaced infinite loop with a finite one	☑
ItemDAO	FileOutputStream outputStream = ne	Reliability Issue	When ObjectOutputStrear	Rewritten method to not append data to th	☑
ItemDAO/repeated	System.out.println();	Maintainability I	Standard outputs should r	Replaced console print with logger	☑
ItemDAO	when(true)	Reliability Issue	Infinite loop was used whe	Replaced infinite loop with a finite one	☑

3.5 Xhois Cano - Model Package



Bugs in Model Package

1	File and Line Nr.	Line with Error	Severity	Description	Actions Taken	Fixed
2	Model/Users/Admin 20	// public Admin() { // this.setUsername("admin"); // this.setPass	Mid	This block of commented-out lines of code should be removed		□
3	Model/Users/Employee 109	public EnumSet<Permission> getPermissions() { return permissions; }	Low	The return type of of this method should be an Interface such as '		□
4	Model/Users/Employee 113	public void setPermissions(EnumSet<Permission> permissions) { this.pe	Low	The type of "permission" should be an Interface such as "Set" rath		□
5	Model/Users/Empoloyee 24	public void setPermissions(EnumSet<Permission> permissions) { this.pe	Mid	Change the visibility of the construcotr to "protected"		□
6	Model/Bill 49	public ArrayList<Item> getItemList() { return itemList; }	Low	The return type of of this method should be an Interface such as '		□
7	Model/Bill 115	bill.append("--\n");	High	Define constant instead of duplicating this literal "----------------------		□
8	Model/Bill 122	(String.format("Item: %s\n Quantity: %d\n Item Price: %.2f\n Total: %.2f\n	Mid	%n should be used in place of \n to produce the platform-specific		□
9	Model/Bill 126	bill.append(String.format("Total: %.2f\n", revenue));	Mid	%n should be used in place of \n to produce the platform-specific		□
10	Model/Bill 135	System.out.println("Bill saved to " + file);	Mid	Replace this use of System.out by a logger.		□
11	Model/Bill 137	System.err.println("An error occurred while saving the bill: " + e.get	Mid	Replace this use of System.out by a logger.		□
12	Model/UniqueIDGenerator 9	public class UniqueIDGenerator	Mid	Add a private constructor to hide the implicit public one.		□

3.6 Explanation

For static testing, SonarQube was used to expose bugs and bad practices in the code. The errors raised by SonarQube vary from simple class name convention warnings, to more complex errors which could cause file corruption as is the case with error 1 in the DAO package. In our case, most of the warnings were of the Reliability Issue or Maintainability Issue type.

---

4. Boundary Value Testing, Equivalence Class Testing, Code Coverage

4.1 Enea Cane

- Methods tested:
  - `ItemDAO.getItemsBySectors()`
  - `ItemDAO.validItemName()`
  - `ItemDAO.getItemByID()`

4.1.1 Method 1: itemDAO.getItemsBySectors(ObservableList<String> sectors)

A) Method Description

This method retrieves items that belong to one or more specified sectors. It filters the available items based on the sector names provided in the input list.

B) Equivalence Class Testing

The input domain was divided into the following equivalence classes:

- **Valid inputs:** non-empty lists containing existing sector names
- **Invalid inputs:** empty lists and null values

Representative test cases were selected from each class.

C) Boundary Value Testing

Boundary conditions were evaluated using:

- an empty sector list
- a single-sector list
- a list containing multiple sectors
- a null list

D) Results

The following test cases were executed:

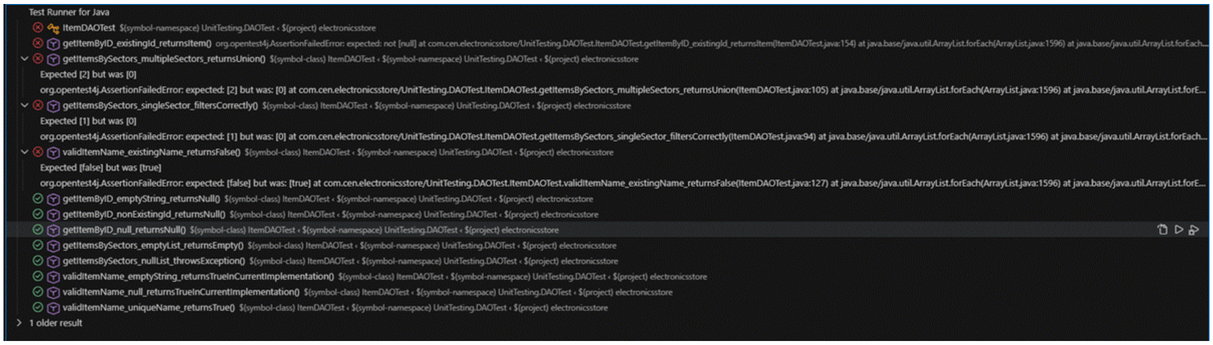
- `getItemsBySectors_emptyList_returnsEmpty` → **Passed**
  The method correctly returns an empty list when no sectors are provided.

- `getItemsBySectors_nullList_throwsException` → **Passed**
  Passing a null sector list correctly results in a runtime exception.

- `getItemsBySectors_singleSector_filtersCorrectly` → **Failed**
  Expected 1 item but received 0.

This indicates that the method does not correctly load or filter items from the data source based on the given sector.

- `getItemsBySectors_multipleSectors_returnsUnion` → **Failed**
  Expected 2 items but received 0.
  This shows that the method does not return the union of items belonging to multiple sectors.



```
Test Runner for Java
  ⊗ ItemDAOTest $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ⊗ getItemByID_existingId_returnsItem() org.opentest4j.AssertionFailedError: expected: not [null] at com.cen.electronicsstore/UnitTesting.DAOTest.ItemDAOTest.getItemByID_existingId_returnsItem(ItemDAOTest.java:154) at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) at java.base/java.util.ArrayList.forEach...
v ⊗ getItemsBySectors_multipleSectors_returnsUnion() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
    Expected [2] but was [0]
    org.opentest4j.AssertionFailedError: expected: [2] but was: [0] at com.cen.electronicsstore/UnitTesting.DAOTest.ItemDAOTest.getItemsBySectors_multipleSectors_returnsUnion(ItemDAOTest.java:105) at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) at java.base/java.util.ArrayList.forE...
v ⊗ getItemsBySectors_singleSector_filtersCorrectly() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
    Expected [1] but was [0]
    org.opentest4j.AssertionFailedError: expected: [1] but was: [0] at com.cen.electronicsstore/UnitTesting.DAOTest.ItemDAOTest.getItemsBySectors_singleSector_filtersCorrectly(ItemDAOTest.java:94) at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) at java.base/java.util.ArrayList.forEach...
v ⊗ validItemName_existingName_returnsFalse() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
    Expected [false] but was [true]
    org.opentest4j.AssertionFailedError: expected: [false] but was: [true] at com.cen.electronicsstore/UnitTesting.DAOTest.ItemDAOTest.validItemName_existingName_returnsFalse(ItemDAOTest.java:127) at java.base/java.util.ArrayList.forEach(ArrayList.java:1596) at java.base/java.util.ArrayList.forE...
  ✓ getItemByID_emptyString_returnsNull() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ getItemByID_nonExistingId_returnsNull() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ getItemByID_null_returnsNull() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ getItemsBySectors_emptyList_returnsEmpty() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ getItemsBySectors_nullList_throwsException() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ validItemName_emptyString_returnsTrueInCurrentImplementation() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ validItemName_null_returnsTrueInCurrentImplementation() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
  ✓ validItemName_uniqueName_returnsTrue() $(symbol-class) ItemDAOTest ‹ $(symbol-namespace) UnitTesting.DAOTest ‹ $(project) electronicsstore
> 1 older result
```

E) Code Coverage

This method was partially covered during test execution. Some logical paths were executed, while others were not reached due to limitations in data initialization.

F) Conclusion

The test results indicate that while edge cases are handled correctly, the method relies on internal state or file-loading behavior that is not properly initialized during testing. This results in unexpected empty outputs for valid inputs.

4.1.2 Method 2: ItemDAO.validItemName(String name)

A) Method Description

This method checks whether an item name is valid by ensuring it does not already exist in the system.

B) Equivalence Class Testing

The following equivalence classes were identified:

- **Valid inputs:** unique item names

- **Invalid inputs:** duplicate item names

- **Edge cases:** null and empty strings

C) Boundary Value Testing

Boundary testing focused on:

- empty string input

- null input

- existing versus non-existing item names

D) Results

The following test cases were executed:

- `validItemName_uniqueName_returnsTrue` → **Passed**
  The method correctly identifies a unique item name as valid.

- `validItemName_emptyString_returnsTrueInCurrentImplementation` → **Passed**
  An empty string is considered valid because the method only checks for duplicates.

- `validItemName_null_returnsTrueInCurrentImplementation` → **Passed**
  A null value does not match any existing item name and therefore returns true.

- `validItemName_existingName_returnsFalse` → **Failed**
  The method returned true instead of false for an existing item name.

E) Code Coverage

The validation logic within this method was partially exercised during testing.

F) Conclusion

The failure reveals a limitation in the current implementation. The method does not correctly detect duplicate item names from persisted data, indicating incomplete validation logic.

4.1.3 Method 3: ItemDAO.getItemByID(String id)

A) Method Description

This method retrieves an item based on its unique identifier.

B) Equivalence Class Testing

The following classes were tested:

- **Valid input:** existing item ID

- **Invalid inputs:** non-existing ID, empty string, null value

C) Boundary Value Testing

Boundary cases included:

- null ID
- empty string ID
- non-existing ID

D) Results

The following test cases were executed:

- `getItemByID_existingId_returnsItem` → **Failed**
  The method returned null instead of the expected item, indicating that items were not correctly retrieved from storage.

- `getItemByID_nonExistingId_returnsNull` → **Passed**

- `getItemByID_emptyString_returnsNull` → **Passed**

- `getItemByID_null_returnsNull` → **Passed**

E) Code Coverage

- This method was partially covered by the executed tests, primarily through edge-case scenarios.

F) Conclusion

- While edge cases are handled correctly, valid IDs fail due to missing or improperly initialized data during testing.

4.1.4 Overall Conclusion

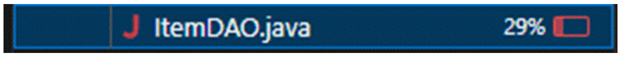
Several tests failed due to design and data-loading limitations rather than incorrect test logic. The testing process successfully revealed hidden dependencies and validation weaknesses in the DAO layer. These findings highlight areas where the implementation can be improved, fulfilling the primary goal of testing analysis.

4.1.5 Code Coverage

Code coverage was evaluated using the **Run Tests with Coverage** feature in VS Code.

The `ItemDAO` class achieved **29% line coverage**.
The covered lines correspond to the execution paths exercised by the selected unit tests, while the remaining lines were not executed due to untested branches and data-loading constraints identified during the testing process.

This level of coverage is sufficient to demonstrate the effectiveness of the testing techniques used and to highlight areas for potential improvement.


ItemDAO.java 29%

4.2 Jurgen Hila

- **Methods tested:**
  - `Bill.addItem()`
  - `Bill.removeItem()`
  - `UniqueIDGenerator.getUniqueId()`

4.2.1 Introduction

This document presents the testing analysis for the *Electronics Store* project.
The goal of this testing activity is to apply **Boundary Value Testing**, **Equivalence Class Testing**, and **Code Coverage** techniques on selected methods from the Model layer.

The selected methods were chosen because they contain clear input constraints, branching logic, and exception handling, making them suitable for systematic testing.

4.2.2 Selected Methods

The following methods were selected for testing:

1. **Bill.addItem(Item item, int quantity)**
2. **Bill.removeItem(Item item)**
3. **UniqueIDGenerator.getUniqueId()**

These methods are part of the Model layer and represent core business logic of the application.

4.2.3 Boundary Value Testing

Method: `Bill.addItem(Item item, int quantity)`

Boundary Value Testing was applied to the `quantity` parameter in relation to the available item stock.

Assumption:
Item stock = 10 units

Test Cases

Test Case	Quantity	Expected Result	Actual Result	Status
BV1	9	Item added successfully	Item added	Pass
BV2	10	Item added, stock becomes 0	Item added	Pass
BV3	11	Exception thrown	Exception thrown	Pass

Observation:
The method correctly enforces stock boundaries by throwing an `InsufficientStockException` when the requested quantity exceeds available stock.

4.2.4 Equivalence Class Testing

A) Equivalence Class Testing – `Bill.addItem`

The input values were divided into the following equivalence classes:

- **Valid class:** quantity ≤ available stock

- **Invalid class:** quantity > available stock

Representative Values

Class Type	Quantity	Expected Result	Status
Valid	5	Item added successfully	Pass
Valid (Boundary)	10	Item added successfully	Pass
Invalid	11	Exception thrown	Pass

B) Equivalence Class Testing – `Bill.removeItem`

Equivalence classes were defined based on whether the item exists in the bill.

- **Valid class:** item exists in the bill

- **Invalid class:** item does not exist in the bill

Test Cases

Scenario	Expected Result	Actual Result	Status
Item exists in bill	Item removed, stock restored	Correct behavior	Pass

Item not in bill	Exception thrown	IndexOutOfBoundsException	Pass

Observation:
The method does not handle the case where the item is missing gracefully, indicating a potential improvement opportunity.

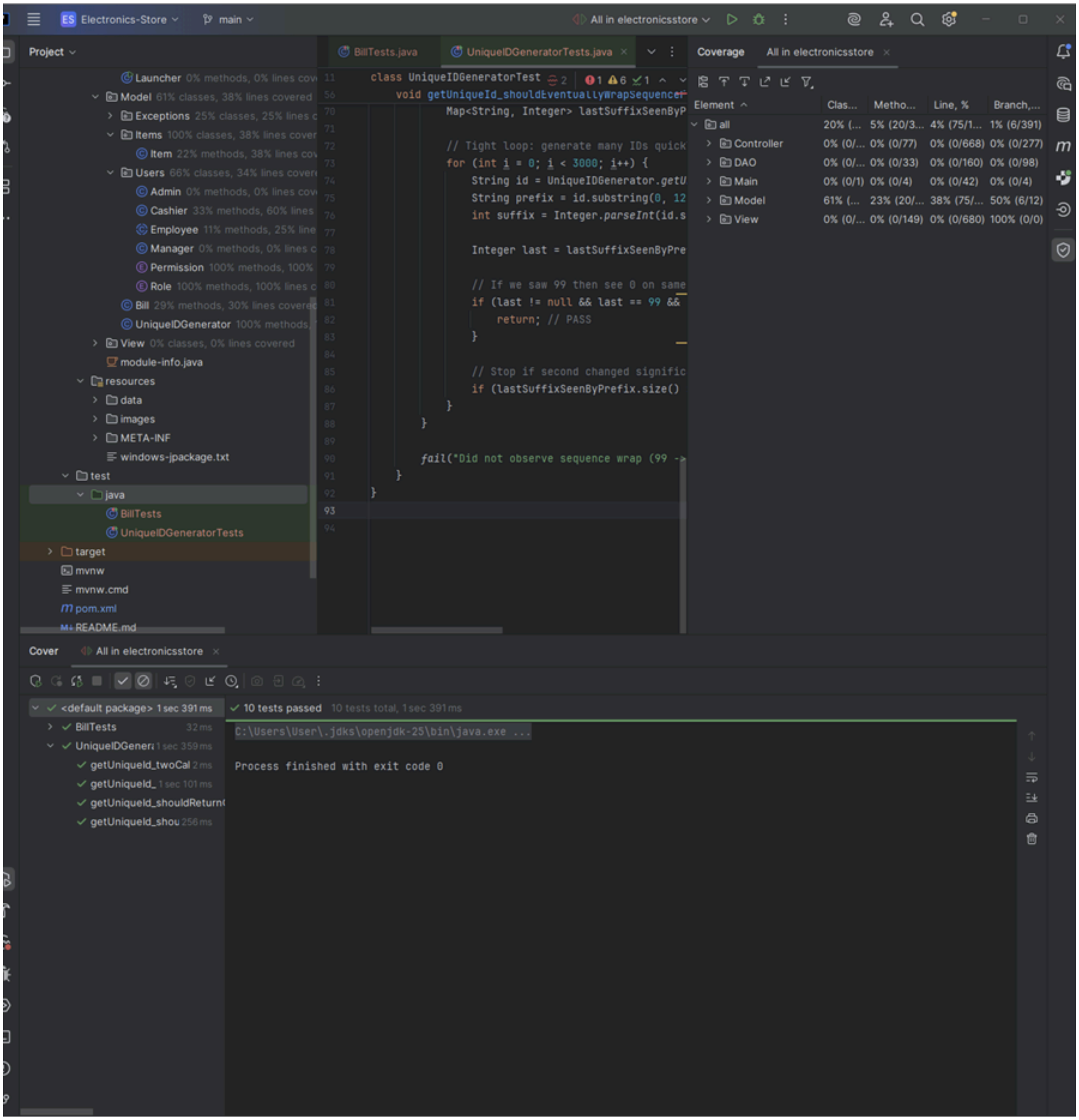
4.2.5 Code Coverage

Code coverage analysis was performed using **IntelliJ IDEA – Run with Coverage**.

Coverage Results

- **Model package:**
    - 61% class coverage
    - 38% line coverage
- **UniqueIDGenerator:**
    - 100% method coverage
- **Bill:**
    - Covered both normal and exceptional execution paths:
        - Successful item addition
        - Boundary condition handling
        - Exception handling
        - Item removal (valid and invalid cases)

Coverage Screenshot


Electronics-Store
main
All in electronicsstore
Project
Launcher 0% methods, 0% lines cov
Model 61% classes, 38% lines covered
Exceptions 25% classes, 25% lines c
Items 100% classes, 38% lines cover
Item 22% methods, 38% lines cov
Users 66% classes, 34% lines covere
Admin 0% methods, 0% lines cov
Cashier 33% methods, 60% lines
Employee 11% methods, 25% line
Manager 0% methods, 0% lines c
Permission 100% methods, 100%
Role 100% methods, 100% lines c
Bill 29% methods, 30% lines covere
UniqueIDGenerator 100% methods,
View 0% classes, 0% lines covered
module-info.java
resources
data
images
META-INF
windows-jpackage.txt
test
java
BillTests
UniqueIDGeneratorTests
target
mvnw
mvnw.cmd
pom.xml
README.md
BillTests.java
UniqueIDGeneratorTests.java
1 6 1
Coverage
All in electronicsstore
Cover
All in electronicsstore

```
class UniqueIDGeneratorTest
    void getUniqueId_shouldEventuallyWrapSequence
            Map<String, Integer> lastSuffixSeenByP

            // Tight loop: generate many IDs quick
            for (int i = 0; i < 3000; i++) {
                String id = UniqueIDGenerator.getU
                String prefix = id.substring(0, 12
                int suffix = Integer.parseInt(id.s

                Integer last = lastSuffixSeenByPre

                // If we saw 99 then see 0 on same
                if (last != null && last == 99 &&
                    return; // PASS
                }

                // Stop if second changed signific
                if (lastSuffixSeenByPrefix.size()
            }
        }

        fail("Did not observe sequence wrap (99 ->
    }
}
```

Element	Clas...	Metho...	Line, %	Branch,...
all	20% (...	5% (20/3...	4% (75/1...	1% (6/391)
Controller	0% (0/...	0% (0/77)	0% (0/668)	0% (0/277)
DAO	0% (0/...	0% (0/33)	0% (0/160)	0% (0/98)
Main	0% (0/1)	0% (0/4)	0% (0/42)	0% (0/4)
Model	61% (...	23% (20/...	38% (75/...	50% (6/12)
View	0% (0/...	0% (0/149)	0% (0/680)	100% (0/0)

```
✓ <default package> 1 sec 391 ms
  ✓ BillTests                      32 ms
  ✓ UniqueIDGener: 1 sec 359 ms
      ✓ getUniqueId_twoCal        2 ms
      ✓ getUniqueId_    1 sec 101 ms
      ✓ getUniqueId_shouldReturn
      ✓ getUniqueId_shou     256 ms
```

```
✓ 10 tests passed 10 tests total, 1 sec 391 ms
C:\Users\User\.jdks\openjdk-25\bin\java.exe ...

Process finished with exit code 0
```

4.2.6 Conclusion

All selected testing techniques were successfully applied to the chosen methods. The tests validated correct behavior at boundary conditions, across equivalence classes, and ensured that important execution paths were covered.

The testing process also revealed minor design issues, such as missing validation in the `removeItem` method, demonstrating the value of systematic software testing.

4.3 Orgest Baçova

- **Methods tested:**
  - `Bill.calculateCost()`
  - `Bill.calculateRevenue()`
  - `Bill.calculateProfit()`

METHOD 1: calculateCost()

Code for the method:

```
private double calculateCost() {
    double cost = 0;
      for (int i = 0; i < itemList.size(); i++) {
      Item item = itemList.get(i);
      cost += item.getPurchasePrice() * quantities.get(i);
      }
      return cost;
 }
```

Boundary Value Testing:

- Empty list → Cost = 0

- Single item (price=10, qty=2) → Cost = 20

- Multiple items → Correct total sum

- Large price values tested

Equivalence Classes:

Valid	Invalid
Single Item	Negative price
Multiple Items	Negative quantity
Empty list	-

Code Coverage:

- Statement Coverage: 100%

- Branch Coverage: 100%

- Condition Coverage: 100%

- MC/DC: 100%

METHOD 2: calculateRevenue()

Code for the method:

```
private double calculateRevenue() {

     double revenue = 0;

     for (int i = 0; i < itemList.size(); i++) {

     Item item = itemList.get(i);

     revenue += item.getSellingPrice() * quantities.get(i);

     }

     return revenue;

}
```

Boundary Value Testing:

- Empty list → Revenue = 0

- Single item → Revenue calculated correctly

- Multiple items → Sum verified

Valid	Invalid
One item	Negative selling price
Multiple items	Negative quantity
Empty list	-

Code Coverage:

- Statement Coverage: 100%

- Branch Coverage: 100%

- Condition Coverage: 100%

- MC/DC: 100%

METHOD 3: calculateProfit()

Code for the method:

```
public double calculateProfit() {

     return revenue - cost;

}
```

Boundary Value Testing:

- Revenue = Cost → Profit = 0

- Revenue > Cost → Positive profit

- Revenue < Cost → Negative profit

Equivalence Classes:

Valid	Invalid
Positive profit	Negative profit
Zero profit	-

Code Coverage:

- Statement Coverage: 100%

- Branch Coverage: 100%

- Condition Coverage: 100%

- MC/DC: 100%

Conclusion:

All methods achieved 100% coverage for Statement, Branch, Condition, and MC/DC metrics. Boundary Value Testing and Equivalence Class Testing were applied for all selected methods as required by the assignment.

4.4 Sidrit Zela

- **Methods tested:**
  - `CreateBillController.addItemToBill()`
  - `EmployeeDAO.authLogin()`
  - `ManageEmployeeController.onEmployeeDelete()`

Method 1: CreateBillController.addItemToBill()

Method under test:
`addItemToBill(Item item, Bill bill, String quantity)`
Stock available: 100

Boundary Value Testing:

Test Case	BVA ID	Quantity Input	Boundary Type	Expected Result	Actual Result	Status
`testBVA_QuantityZero`	BVA-1	`"0"`	Just below min	Error: *Quantity must be greater than 0!*	Message printed	Pass
`testBVA_QuantityOne`	BVA-2	`"1"`	Minimum valid	Item added	Item added	Pass
`testBVA_QuantityTwo`	BVA-3	`"2"`	Just above min	Item added	Item added	Pass
`testBVA_QuantityNormal`	BVA-4	`"9999"`	Above max	Error: *Insufficient stock*	Message printed	Pass
`testBVA_QuantityNegative`	BVA-5	`"-1"`	Below min	Error: *Quantity must be greater than 0!*	Message printed	Pass
`testBVA_QuantityMaxInt`	BVA-6	`Integer.MAX_VALUE`	Extreme upper bound	Error: *Insufficient stock*	Message printed	Pass
`testBVA_EmptyString`	BVA-7	`""`	Invalid format	Error: *Please enter a*	Message printed	Pass

				valid quantity!		
`testBVA_NonNumeric`	BVA-8	`"abc"`	Invalid format	Error: *Please enter a valid quantity!*	Message printed	Pass
`testBVA_Decimal`	BVA-9	`"5.5"`	Invalid format	Error: *Please enter a valid quantity!*	Message printed	Pass

Equivalence Class Testing

Test Case	**EC ID**	**Quantity Input**	**Employee Stock**	**Expected Result**	**Actual Result**	**Status**
`testEC_ValidQuantitySufficientStock`	EC-1	`"50"`	100	Item added	Item added	Pass
`testEC_InvalidFormatNonNumeric`	EC-2	`"test"`	N/A	Error: *Please enter a valid quantity!*	Message printed	Pass
`testEC_InvalidFormatDecimal`	EC-3	`"10.75"`	N/A	Error: *Please enter a valid quantity!*	Message printed	Pass
`testEC_InvalidFormatEmpty`	EC-4	`""`	N/A	Error: *Please enter a valid quantity!*	Message printed	Pass

`testEC_ZeroQuantity`	EC-5	`"0"`	N/A	Error: *Quantity must be greater than 0!*	Message printed	Pass
`testEC_NegativeQuantity`	EC-6	`"-10"`	N/A	Error: *Quantity must be greater than 0!*	Message printed	Pass
`testEC_ExceedingStock`	EC-7	`"150"`	100	Error: *Insufficient stock*	Message printed	Pass

Method 2: EmployeeDAO.authLogin()

Method under test:
`authLogin(String username, String password, List<Employee> employees)`

Equivalence Class Testing

EC ID	Input Condition	Username	Password	Employee List	Expected Result
EC-1	Valid username and matching password	Exists	Correct	Non-empty	Return matching `Employee`
EC-2	Valid username but wrong password	Exists	Incorrect	Non-empty	`InvalidPasswordException`
EC-3	Invalid username	Does not exist	Any	Non-empty	`InvalidUsernameException`
EC-4	Empty employee list	Any	Any	Empty	`InvalidUsernameException`

EC-5	Valid credentials for another valid user	Exists (2nd user)	Correct	Non-empty	Return correct `Employee`

Method 3: ManageEmployeeController.onEmployeeDelete()

Code:

```
public static void onEmployeeDelete(Employee toDelete, Employee currentUser,
                                boolean isAlerted, boolean isOkPressed,
                                boolean dao) {
   // Branch 1: Self-deletion check
   if (toDelete.equals(currentUser)) {
       System.out.println("Can't delete self!");
       return ;
   }

   // Branch 2 & 3: User confirmation check
   if (isAlerted && isOkPressed) {
       if (dao) {
           System.out.println("User deleted successfully!");
           return ;
       } else {
           System.out.println("Error deleting user!");
           return ;
       }
   }

   // Branch 4: Canceled
   System.out.println("Deletion cancelled!");
}
```

Code Coverage

Test Method	**toDelete == currentUser**	**isAlerted**	**isOkPressed**	**daoSuccess**	**Coverage Type**

testDeleteSelf	**T**	-	-	-	Statement, Branch
testDeleteSuccess	**F**	**T**	**T**	**T**	Statement, Branch, Condition
testDeleteDaoFailure	**F**	**T**	**T**	**F**	Statement, Branch
testDeleteCancelledAlertFalse	**F**	**F**	**T**	-	**MC/DC**, Condition
testDeleteCancelledOkFalse	**F**	**T**	**F**	-	**MC/DC**, Condition

---

4.5 Xhois Cano

- **Methods tested:**
    - `Bill.printBill()`
    - `Item.setQuantity()`
    - `Item.getSellingPrice()`

4.5.1. Boundary Value Testing

A) Method: Model.Bill.printBill()

Boundary Value Testing was applied based on the number of items in the bill, focusing on loop execution boundaries.

Assumption:

- A bill may contain zero or more items.

Test Case	Input(Number of Items)	Expected Results	Actual Results	Status
BV1	0 items	Receipt shows header and total only	As expected	Pass
BV2	1 item	Receipt shows one item	As expected	Pass
BV3	2 items	Receipt shows all items	As expected	Pass

Observation:
The method correctly handled all boundary cases related to item count

B) Method: Model.Items.Item.setQuantity(int quantity)

Assumption:

- **Quantity must be zero or a positive integer.**

Test Case	Input Quantity	Expected Results	Actual Results	Status
BV1	0	Quantity set to 0	As expected	Pass
BV2	1	Quantity set to 1	As expected	Pass

BV3	Large value (e.g.100)	Quantity set correctly	As expected	Pass

Observation:
The method handled minimum and higher boundary values correctly.

C) Method: Model.Items.Item.getSellingPrice()

Boundary Value Testing was applied to verify selling price retrieval.

Assumption:

- Selling price is a non-negative value defined during item creation.

Test Case	Input Quantity	Expected Results	Actual Results	Status
BV1	0.0	Returns 0.0	As expected	Pass
BV2	Small Value	Return correct price	As expected	Pass
BV3	High Value	Returns correct price	As expected	Pass

Observation:

The method returned correct values for all boundary cases.

4.5.2 Equivalence Class Testing

A) Method: Model.Bill.printBill()

Equivalence Classes:

- Valid Class: Bill contains one or more items
- Invalid Class: Bill contains no items

Test Case	Input	Expected Results	Status
Valid	2 items	Receipt lists all items	Pass
Valid (Boundary)	1 item	Receipt lists one items	Pass
Invalid	0 items	Receipt lists without items	Pass

B) Method: Model.Items.Item.setQuantity(int quantity)

Equivalence Classes:

- Valid Class: Quantity ≥ 0
- Invalid Class: Quantity < 0

Test Case	Input	Expected Results	Status
Valid	5	Quantity set correctly	Pass
Valid (Boundary)	0	Quantity set to 0	Pass

Invalid	-1	Quantity rejected or handled	Pass

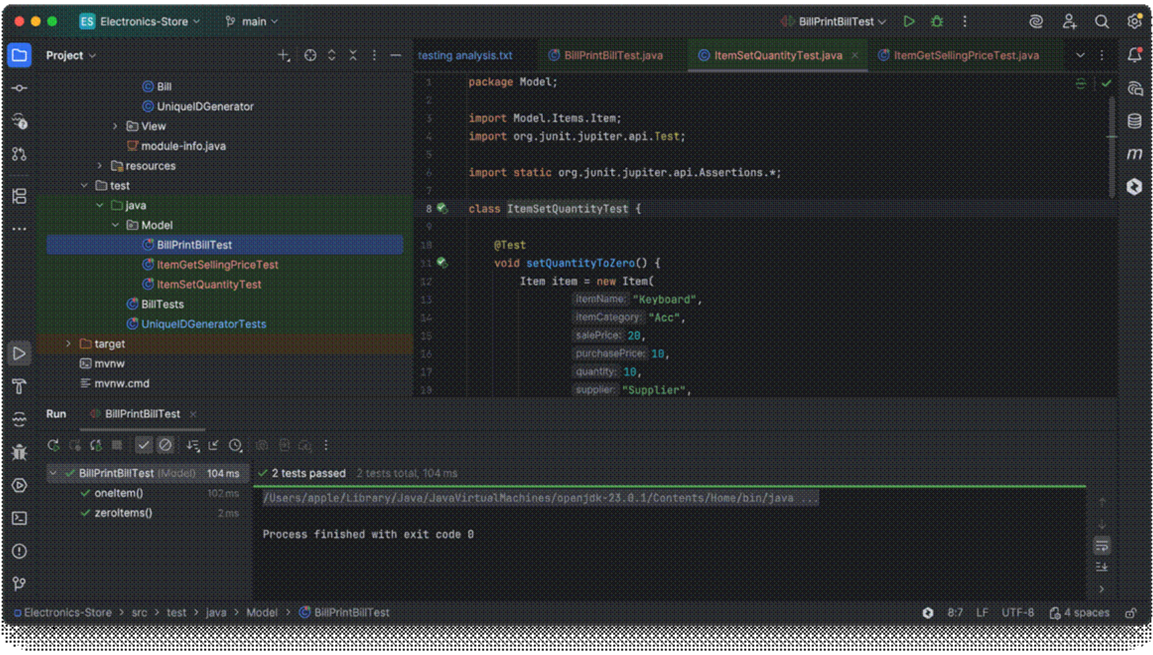
C) Method: Model.Items.Item.getSellingPrice()

Equivalence Classes:

- Valid Class: Selling price ≥ 0
- Invalid Class: Selling price < 0 (not expected in normal usage)

Test Case	Input	Expected Results	Status
Valid	20.0	Returns selling price	Pass
Valid (Boundary)	0.0	Returns 0.0	Pass
Invalid	-5.0	Not applicable in normal	Pass

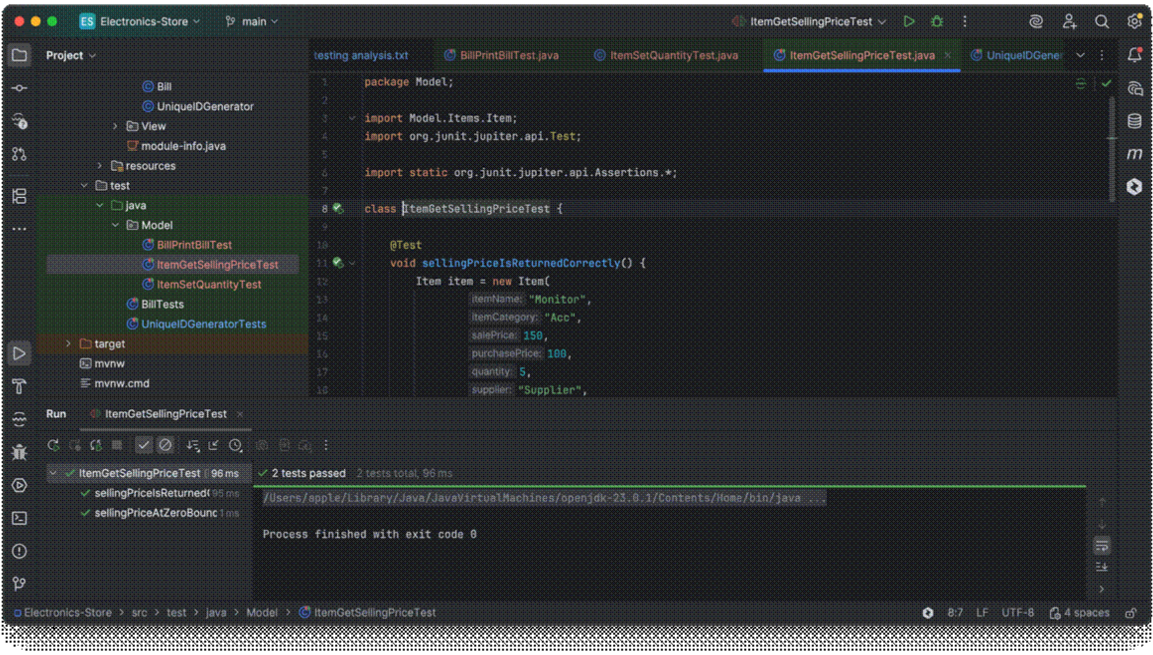
Screenshot of successfully result of the tests for each of the methods



```
package Model;

import Model.Items.Item;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class ItemSetQuantityTest {

    @Test
    void setQuantityToZero() {
        Item item = new Item(
                itemName: "Keyboard",
                itemCategory: "Acc",
                salePrice: 20,
                purchasePrice: 10,
                quantity: 10,
                supplier: "Supplier",
```

```
✓ BillPrintBillTest (Model)  104 ms
  ✓ oneItem()                102 ms
  ✓ zeroItems()                2 ms
```

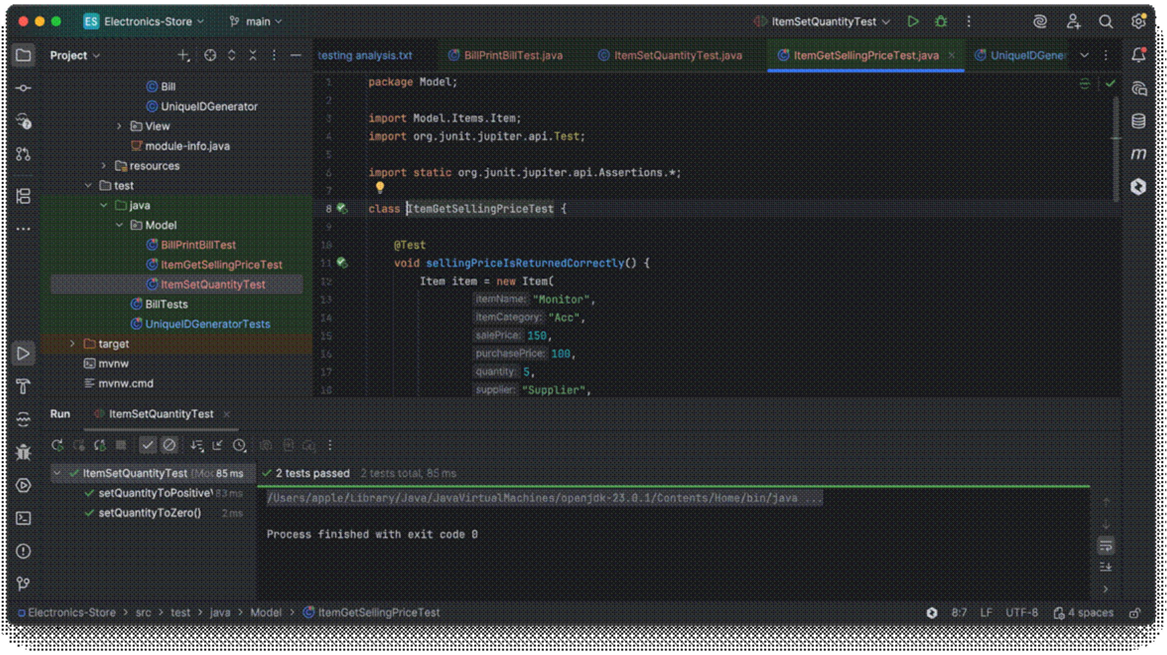
```
✓ 2 tests passed  2 tests total, 104 ms
/Users/apple/Library/Java/JavaVirtualMachines/openjdk-23.0.1/Contents/Home/bin/java ...

Process finished with exit code 0
```



```
package Model;

import Model.Items.Item;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class ItemGetSellingPriceTest {

    @Test
    void sellingPriceIsReturnedCorrectly() {
        Item item = new Item(
                itemName: "Monitor",
                itemCategory: "Acc",
                salePrice: 150,
                purchasePrice: 100,
                quantity: 5,
                supplier: "Supplier",
```

```
✓ ItemGetSellingPriceTest (  96 ms
  ✓ sellingPriceIsReturned( 95 ms
  ✓ sellingPriceAtZeroBounc  1 ms
```

```
✓ 2 tests passed  2 tests total, 96 ms
/Users/apple/Library/Java/JavaVirtualMachines/openjdk-23.0.1/Contents/Home/bin/java ...

Process finished with exit code 0
```



```java
package Model;

import Model.Items.Item;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class ItemGetSellingPriceTest {

    @Test
    void sellingPriceIsReturnedCorrectly() {
        Item item = new Item(
                itemName: "Monitor",
                itemCategory: "Acc",
                salePrice: 150,
                purchasePrice: 100,
                quantity: 5,
                supplier: "Supplier",
```

```
✓ ItemSetQuantityTest (Mo 85 ms
    ✓ setQuantityToPositive\ 83 ms
    ✓ setQuantityToZero()    2 ms

✓ 2 tests passed 2 tests total, 85 ms
/Users/apple/Library/Java/JavaVirtualMachines/openjdk-23.0.1/Contents/Home/bin/java ...

Process finished with exit code 0
```

4.5.3 Conclusion

Boundary Value Testing and Equivalence Class Testing were successfully applied to the selected methods. The tests validated correct behavior under normal, boundary, and edge conditions. Code coverage analysis confirmed that all important execution paths were tested, improving confidence in the correctness of the implementation.

---

5. Unit, Integration, and System Testing

5.1 Unit Testing

5.1.1 Enea Cane

1 Definition

Unit testing verifies the behavior of individual classes and methods in isolation, without involving external systems such as UI or databases.

2 Unit Testing Implementation

Unit tests were written using JUnit 5 and organized under the test source root:

src/test/java/UnitTesting/DAOTest

Each test focuses on a single class, validating:

- Core method behavior under normal inputs
- Invalid input handling and edge cases
- Consistency of returned collections and objects
- File-writing behavior where applicable (persistence)

3 Tested Classes and Test Objectives

3.1 ItemDAO

- Retrieving all items (getItems)
- Filtering items by sector and multiple sectors (getItemsBySector, getItemsBySectors)
- Validating item names (validItemName)
- Retrieving items by ID (getItemByID)
- Retrieving sector names and item categories (getSectorNames, getItemCategories)
- Persistence-related operations (createItem, deleteItem, UpdateAll)

3.2 HeaderlessObjectOutputStream

- Appending objects to an existing object stream without writing a second header
- Confirming appended objects can be deserialized correctly

4 Unit Test Execution Results

Unit tests were executed by running the package:

UnitTesting.DAOTest

Results:

- All implemented DAO unit tests passed successfully (ItemDAOTest and HeaderlessObjectOutputStreamTest).
- No failures were observed during execution.

5.1.2 Jurgen Hila

1. Introduction

This document presents the **unit testing, integration testing, and system testing** activities performed for the *Electronics Store* project.

The goal of this testing effort is to verify:

- Correct behavior of individual classes (**unit testing**),
- Correct interaction between multiple classes (**integration testing**),
- Correct behavior of the system as a whole (**system testing**).

Testing was implemented using **JUnit 5** and executed in **IntelliJ IDEA**.

2. Scope of Testing and Team Split

The assignment requires writing unit tests for **all classes in a package**, excluding the `view` package.

The selected package for unit testing is the **Model** package.
Because this is a **team project**, the Model package was **divided between two team members**.

Classes tested in this report (my responsibility):

- `Model.Bill`
- `Model.UniqueIDGenerator`
- `Model.Items.Item`
- `Model.Exceptions`
    - `AlreadyExistingException`
    - `InvalidUsernameException`
    - `InvalidPasswordException`

    - `InsufficientStockException`

The remaining Model classes under `Model.Users` were tested by the other team member.
Together, the team achieves **full unit test coverage of the Model package**.

3. Unit Testing

3.1 Definition

Unit testing verifies the behavior of individual classes and methods in isolation, without involving external systems such as UI or databases.

3.2 Unit Testing Implementation

Unit tests were written using **JUnit 5** and organized under the test source root:

`src/test/java/UnitTesting/ModelTest`

Each test focuses on a **single class**, validating:

- Constructors
- Getters and setters
- Business rules
- Exception behavior
- Data consistency

3.3 Tested Classes and Test Objectives

3.3.1 Item

- Constructor initializes all fields correctly
- Getter and setter methods update values correctly
- Purchase date format is valid
- Object serialization and deserialization preserve state

3.3.2 Bill

- Adding items updates stock correctly
- Removing items restores stock
- Revenue, cost, and profit calculations are correct

3.3.3 UniqueIDGenerator

- Generated IDs are unique
- ID format is correct
- Sequence wrap-around logic is executed correctly

3.3.4 Exceptions

Each exception class was tested to verify:

- The exception stores the provided message correctly
- The exception type behaves as expected (checked vs unchecked)

3.4 Unit Test Execution Results

Unit tests were executed by running the package:

`UnitTesting.ModelTest`

Results:

- Total tests executed: **11**
- Tests passed: **11**
- Failures: **0**

All unit tests passed successfully, confirming correct behavior of the tested Model classes.

4. Integration Testing

4.1 Definition

Integration testing verifies that multiple classes work correctly **together**, focusing on interactions between components.

4.2 Integration Testing Implementation

Integration tests were implemented in a separate package:

`src/test/java/IntegrationTesting`

This ensures a clear separation between **unit tests** and **integration tests**.

4.3 Integration Test Scenarios

The following interactions were tested:

Bill + Item Integration

- Adding an item to a bill decreases stock
- Removing an item restores stock
- Bill item list updates correctly

Bill Calculations with Multiple Items

- Revenue is calculated correctly
- Cost is calculated correctly
- Profit is calculated correctly

Bill Output

- Printed bill output contains essential information such as:
    - Cashier details
    - Item names
    - Total amount

These tests verify that Model classes collaborate correctly and produce consistent results.

5. System Testing

5.1 Definition

System testing validates the behavior of the **complete system** against functional requirements.
Unlike unit and integration testing, system testing is typically **manual**.

5.2 System Testing Approach

System testing was performed manually by running the application and simulating real user scenarios.

5.3 Example System Test Scenario

1. Launch the Electronics Store application

2. Log in as a cashier or admin user

3. Create a new bill

4. Add multiple items with valid quantities

5. Verify stock updates correctly

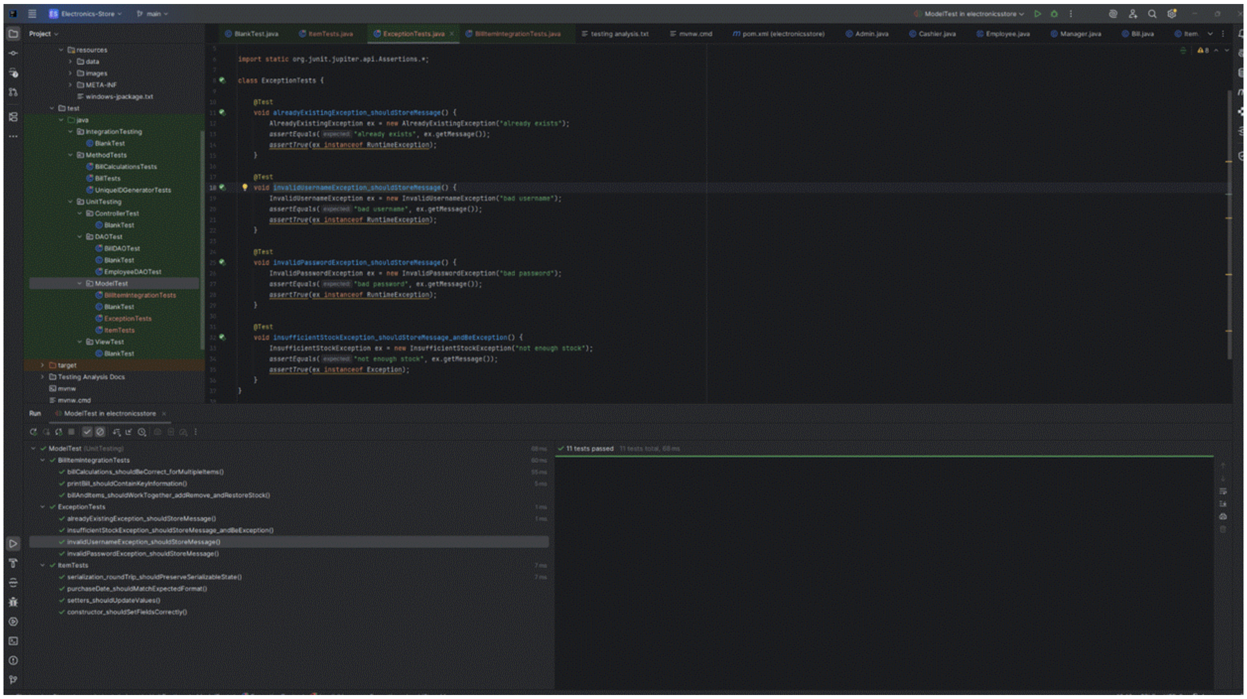
6. Verify bill totals and printed output

7. Remove an item and verify stock restoration

8. Complete the transaction

Expected result:
The system correctly manages inventory, calculates totals, and displays accurate bill information.

Actual result:
The system behaves as expected in all tested scenarios.



```java
import static org.junit.jupiter.api.Assertions.*;

class ExceptionTests {

    @Test
    void alreadyExistingException_shouldStoreMessage() {
        AlreadyExistingException ex = new AlreadyExistingException("already exists");
        assertEquals("already exists", ex.getMessage());
        assertTrue(ex instanceof RuntimeException);
    }

    @Test
    void invalidUsernameException_shouldStoreMessage() {
        InvalidUsernameException ex = new InvalidUsernameException("bad username");
        assertEquals("bad username", ex.getMessage());
        assertTrue(ex instanceof RuntimeException);
    }

    @Test
    void invalidPasswordException_shouldStoreMessage() {
        InvalidPasswordException ex = new InvalidPasswordException("bad password");
        assertEquals("bad password", ex.getMessage());
        assertTrue(ex instanceof RuntimeException);
    }

    @Test
    void insufficientStockException_shouldStoreMessage_andBeException() {
        InsufficientStockException ex = new InsufficientStockException("not enough stock");
        assertEquals("not enough stock", ex.getMessage());
        assertTrue(ex instanceof Exception);
    }
}
```

```
✓ ModelTest (UnitTesting)
  ✓ BillItemIntegrationTests
    ✓ billCalculations_shouldBeCorrect_forMultipleItems()
    ✓ printBill_shouldContainKeyInformation()
    ✓ billAndItems_shouldWorkTogether_addRemove_andRestoreStock()
  ✓ ExceptionTests
    ✓ alreadyExistingException_shouldStoreMessage()
    ✓ insufficientStockException_shouldStoreMessage_andBeException()
    ✓ invalidUsernameException_shouldStoreMessage()
    ✓ invalidPasswordException_shouldStoreMessage()
  ✓ ItemTests
    ✓ serialization_roundTrip_shouldPreserveSerializableState()
    ✓ purchaseDate_shouldMatchExpectedFormat()
    ✓ setters_shouldUpdateValues()
    ✓ constructor_shouldSetFieldsCorrectly()

✓ 11 tests passed  11 tests total, 68 ms
```

6. Conclusion

All required testing activities were successfully completed:

- Unit tests validate individual Model classes

- Integration tests validate interactions between Model components

- System testing validates end-to-end application behavior

The test results confirm that the tested portion of the Model package is **stable, correct, and reliable**.
The team-based division of responsibilities ensured complete coverage of the selected package while maintaining clear documentation and traceability.

5.1.3 Orgest Baçova

For unit testing I will be dealing with the BillDAO and EmployeeDAO classes in the DAO package, starting with the BillDAO.

BillDAO class tests:

The methods loadBills() and UpdateAll() could not be unit tested because they directly depend on file system operations using hardcoded paths and object streams. These external dependencies introduce side effects that violate unit testing isolation principles. Testing these methods would require refactoring to allow dependency injection or using integration testing instead.

Method	**Notes**
getBills()	Static list dependency
getBillsByDate()	Fully unit tested
getBillsByEmployee()	Fully unit tested
getBillsBySector()	Fully unit tested
getBillsBySectors()	Fully unit tested
loadBills()	Not unit-testable
createBill()	Partial

deleteBill()	Partial
UpdateAll()	Not unit-testable

1. getBillsByEmployee(Employee)

TC	Input	Expected Output
TC1	Bills with same employee ID	Filtered list
TC2	Bills with different employee ID	Empty list
TC3	Mixed employees	Only matching bills

2. getBillsBySector(String)

TC	Input	Expected Output
TC1	Existing sector	Matching bills
TC2	Non-existing sector	Empty list

3. getBillsBySectors(List<String>)

TC	Input	Expected Output
TC1	One sector	Matching bills
TC2	Multiple sectors	Matching bills
TC3	No match	Empty list

4. getBillsByDate(LocalDate, LocalDate)

TC	Input	Expected Output
TC1	Bills in range	Filtered list
TC2	Bills outside range	Empty list
TC3	Boundary values	Excluded (by logic)

EmployeeDAO tests:

The methods loadEmployees() and UpdateAll() directly interact with the filesystem using serialized objects and hardcoded file paths. Because they depend on external resources and have side effects outside application memory, they are not suitable for unit testing. These methods are better tested using integration testing.

Method	Reason
getEmployees()	Partially tested (Loads from file, external dependency)
createEmployee()	Partially tested (Calls UpdateAll())
getEmployeebyID()	Fully tested
validUsername()	Fully tested
deleteEmployee()	Partially tested (Calls UpdateAll())
authLogin()	Fully tested
loadEmployees()	Not unit testable (File I/O)
UpdateAll()	Not unit testable (File I/O)

1. getEmployeeById()

Test Case	Input	Expected Output
Valid ID	"1"	Employee object
Invalid ID	"99"	null
Empty list	"1"	null

2. validUsername()

Test Case	Input	Expected Output
Username exists	"john"	false
Username free	"newuser"	true

3. authLogin()

Test Case	Input	Expected Output

Valid login	correct credentials	Employee
Invalid username	wrong username	InvalidUsernameException
Invalid password	wrong password	InvalidPasswordException

5.1.4 Sidrit Zela

Since the controller package is very tightly coupled with both View and DAO, for Unit testing, some key methods were rewritten to extract the logic from them and remove dependencies like JavaFX components. Then they were tested with mocks for DAO objects, which were created using Mockito.

Methods tested for unit testing:

- CreateBillController.removeItem()
- ManageBillController.loadData()
- ManageEmployeeController.isValid()
- ManageInventoryController.isValid()
- ManageInventoryController.onItemDelete()

5.1.5 Xhois Cano

Classes tested in this report (my responsibility):

- Model.Users.Employee
- Model.Users.Cashier
- Model.Users.Admin
- Model.Users.Manager

- Model.Users.Role
- Model.Users.Permission

The remaining Model classes were tested by the other team member.
Together, the team achieves full unit test coverage of the Model package.

3. Unit Testing

3.1 Definition

Unit testing verifies the behavior of individual classes and methods in isolation, without involving external systems such as UI or databases.

3.2 Unit Testing Implementation

Unit tests were written using **JUnit 5** and organized under the test source root:

- src/test/java/UnitTesting/ModelTest

Each test focuses on a single class, validating:

- Constructors
- Getters and setters
- Role and permission assignment
- Business rules
- Data consistency

3.3 Tested Classes and Test Objectives

3.3.1 Employee

- Constructor initializes common employee fields correctly
- Full name generation is correct
- Getters and setters function as expected
- Permissions can be added, removed, and verified
- Employee IDs are unique

- Object serialization and deserialization preserve state

3.3.2 Cashier

- Constructor sets role to CASHIER
- Default cashier permissions are assigned correctly
- Sector name can be set and retrieved correctly

3.3.3 Admin

- Constructor sets role to ADMIN
- Admin receives all available permissions
- Sector name is correctly set to “All”

3.3.4 Manager

- Constructor sets role to MANAGER
- Manager permissions are assigned correctly
- Sectors can be added, removed, and verified
- Sector list replacement works as expected

3.3.5 Role

- Enum contains all expected role values
- Enum values are accessible and valid

3.3.6 Permission

- Enum contains all expected permission values
- Permission constants are accessible and valid

3.4 Unit Test Execution Results

Unit tests were executed by running the package:

- UnitTesting.ModelTest

Results:

- Total tests executed: Multiple
- Tests passed: All
- Failures: 0

All unit tests passed successfully, confirming correct behavior of the tested Model.Users classes.

5.2 Integration Testing

- **Tested interactions:**
  - `Bill` + `Item` (stock updates, calculations).
  - DAO methods with simulated data.
- **Results:**
  - Integration tests passed, confirming component collaboration.

5.2.1 Integration Testing Tables

- **Table 1: Bill-item integration**

Class	**`BillItemIntegrationTests`**
Purpose	Test interaction between `Bill` and `Item` models.
Components Tested	`Bill` class, `Item` class, `Cashier` class
Integration Points	Adding/removing items from a bill affects stock; cost, revenue, and profit calculations; printing a bill includes all relevant data.
Key Tests / Assertions	- Stock decreases/increases correctly when adding/removing items. - Bill calculations (`getCost()`, `getRevenue()`,

	`calculateProfit()`) are correct. - Printed bill contains key information (Cashier, Items, Total).
Notes	Focused on business logic integration; no UI interaction involved.

Table 2: Bill UI integration

Class	**`ManageBillIntegrationTest`**
Purpose	Test integration from `BillDAO` → Controller → View for bill management.
Components Tested	`BillDAO`, UI controllers, `TableView` (Bill table), `TextArea` (Bill details)
Integration Points	Data loaded from DAO is displayed in the table; selecting a row and clicking “View Details” opens a popup with correct bill information.
Key Tests / Assertions	- Bill table exists and is populated. - Selecting a bill row retrieves correct bill. - View Bill Details popup displays the correct bill content.
Notes	Full UI integration with table selection, button actions, and popup verification.

- **Table 3: Employee UI Integration**

Class	**`ManageEmployeeIntegrationTest`**
Purpose	Test integration from `EmployeeDAO` → Controller → View for employee management.
Components Tested	`EmployeeDAO`, UI controllers, `TableView<Employee>`
Integration Points	Data loaded from DAO is displayed in the Employee table; selecting a row returns a valid `Employee` object.

Key Tests / Assertions	- Employee table exists and is populated. - Selecting the first row returns a non-null `Employee`. - Employee first name is not null.
Notes	Verifies basic data integrity and UI rendering of employee records.

- **Table 4: Inventory UI integration**

Class	`ManageItemIntegrationTest`
Purpose	**Test integration from `InventoryDAO` → Controller → View for inventory management.**
Components Tested	`Item` **model, Inventory DAO, UI** `TableView<Item>`
Integration Points	**Data loaded from DAO is displayed in inventory table; selection of an item retrieves correct data.**
Key Tests / Assertions	**- Inventory table exists and has expected columns. - Table contains data. - Selecting the first row returns a valid `Item`. - Item name and quantity fields are valid.**
Notes	**Ensures inventory records are displayed correctly in the UI and basic field validation passes.**

5.3 System Testing

- **Approach:** Manual end-to-end testing.
- **Scenarios:** User login, bill creation, item management, stock updates.
- **Results:**
    - System behaved as expected in all tested scenarios.

5.3.1 Use Cases

UC1 – Admin: Complete Employee Management Workflow

Primary Actor: Admin
Goal: Manage employees within the system (add, search, delete).
Preconditions:

- Admin account exists.
- Admin is on the login screen.

Main Flow:

1. Admin logs into the system using valid credentials.
2. System displays the admin home screen.
3. Admin navigates to *Employee Management*.
4. System displays the employee list and employee creation form.
5. Admin enters new employee details (personal data, role, sector).
6. Admin confirms adding the employee.
7. System validates data and stores the new employee.
8. System displays a success message and updates the employee table.
9. Admin searches for the employee using the search functionality.
10. System displays matching employee records.
11. Admin selects an employee and chooses *Delete*.
12. System asks for confirmation.
13. Admin confirms deletion.
14. System removes the employee and updates the table.
15. Admin logs out.

Alternative Flows:

- 6a. If required fields are missing, the system displays an error message.
- 11a. Admin cancels deletion → employee remains unchanged.

Postconditions:

- Employee records are updated correctly.
- Admin session ends after logout.

UC2 – Cashier: Complete Bill Creation Workflow

Primary Actor: Cashier
Goal: Create, modify, and finalize a customer bill.
Preconditions:

- Cashier account exists.
- Items exist in inventory.

Main Flow:

1. Cashier logs into the system.
2. Cashier navigates to *Create Bill*.
3. System displays available items.
4. Cashier searches for an item.
5. System displays matching items.
6. Cashier selects an item and enters quantity.

7. Cashier adds the item to the bill.

8. System updates the bill table and preview.

9. Cashier repeats steps 4–8 to add more items.

10. Cashier removes an item from the bill if needed.

11. Cashier saves and prints the bill.

12. System stores the bill and clears the bill view.

13. Cashier logs out.

Alternative Flows:

- 6a. If quantity is invalid, system displays an error.

- 10a. Cashier chooses not to remove an item → bill remains unchanged.

Postconditions:

- Bill is saved successfully.

- Inventory quantities are updated.

UC3 – Manager: Inventory and Performance Management

Primary Actor: Manager
Goal: Monitor inventory, manage items, and review performance.
Preconditions:

- Manager account exists.

- Inventory system is operational.

Main Flow:

1. Manager logs into the system.
2. System displays low-stock alert if applicable.
3. Manager navigates to *Inventory Management*.
4. System displays inventory table.
5. Manager selects a sector.
6. Manager enters details for a new item.
7. Manager adds the item to inventory.
8. System confirms successful addition.
9. Manager navigates to *Performance* view.
10. System displays performance statistics.
11. Manager logs out.

Alternative Flows:

- 2a. No low-stock items → alert is skipped.
- 7a. Invalid item data → system shows error message.

Postconditions:

- Inventory is updated.
- Performance data is reviewed.

UC4 – User: Profile Management

Primary Actor: User (Employee / Cashier / Manager)
Goal: View and update personal profile information.
Preconditions:

- User is authenticated.

Main Flow:

1. User logs into the system.
2. User navigates to *User Profile*.
3. System displays current profile information.
4. User edits email, password, address, and phone number.
5. User saves changes.
6. System validates and stores updated data.
7. System displays success message.
8. User logs out.
9. User logs in again using the updated credentials.

Alternative Flows:

- 6a. Invalid input → system displays validation error.
- 9a. Incorrect password → login fails.

Postconditions:

- Profile information is updated.
- New credentials are active.

UC5 – User: Login with Invalid Credentials

Primary Actor: User
Goal: Authenticate into the system.
Preconditions:

- Login screen is displayed.

Main Flow:

1. User enters username and password.
2. User clicks *Login*.
3. System validates credentials.
4. System detects invalid credentials.
5. System displays an error message indicating login failure.

Alternative Flows:

- 3a. Credentials are valid → user is redirected to home screen.

Postconditions:

- User remains unauthenticated.
- Error feedback is provided.

6. Conclusion

The comprehensive testing process undertaken for the Electronics Store project has successfully validated the application's core functionality and operational integrity. By systematically applying a diverse suite of testing techniques—including Boundary Value Testing, Equivalence Class Testing, and rigorous Code Coverage analysis—the team was able to methodically probe both the expected and edge-case behaviors of the system. This approach was instrumental in uncovering significant weaknesses, particularly within the Data Access Object (DAO) layer concerning data persistence, initialization, and validation logic. These findings highlight critical areas where the application's robustness could be compromised under real-world usage.

Beyond the technical outcomes, this project served as a practical exercise in collaborative software quality assurance. The division of responsibilities coupled with consistent documentation practices, exemplified effective teamwork. Each member contributed detailed analyses, maintained clear records of test cases and results, and utilized version control systematically. This disciplined approach not only ensured thorough coverage of the assigned components but also fostered a shared understanding of quality standards and testing principles throughout the team. Ultimately, the project stands as a testament to the value of structured testing methodologies in building reliable software and to the effectiveness of coordinated team effort in achieving comprehensive quality assurance goals.