

Project

Software Testing And Quality Assurance

Part 3

Worked : Antea Koxherri , Flavia Koco, Jagoda Andrrea

Accepts : Professor Ari Gjerazi

Professor Jurgen Cama

***For this project, the team split the testing work by testing type
(Unit / Integration / System).***

Contents

1. Unit Testing.....	3
1. Unit Testing Scope and Approach.....	3
2. Unit-Tested Classes and Methods (Coverage List).....	3
3. Tables for Test Cases.....	5
4. Methods Not Covered by Unit Testing.....	13
BillingController.....	13
InventoryController.....	13
LoginController.....	13
UserManagementController.....	14
ReportController.....	14
FileHandler.....	14
JavaFX View Classes.....	15
Abstract and Trivial Methods.....	15
5. Integration Testing.....	16
5. Integration Testing Scope and Approach.....	16
6. Integration Tests – Grouping and Coverage.....	17
6.1 Group 1: File Persistence Integration Tests.....	17
6.2 Group 2: Inventory and Session Integration Tests.....	18
6.3 Group 3: Billing Integration Tests.....	18
6.4 Group 4: Reporting Integration Tests.....	19
7. Integration Testing – Method-Level Explanation.....	20
7.1 UserFileAuthenticationIntegrationTest.....	20
7.2 SupplierFileIntegrationTest.....	21
7.3 InventoryItemFileIntegrationTest.....	22
7.4 InventoryFileOverwriteIntegrationTest.....	22
7.5 ReadNonExistingFileIntegrationTest.....	23
7.6 InventorySessionState.....	24
7.7 InventoryCategoryIntegrationTest.....	24
7.8 BillingRoleIntegrationTest.....	25
7.9 BillingFlowIntegrationTest.....	26
7.10 ReportFromBillsIntegrationTest.....	26
7.11 ReportExportIntegrationTest.....	27
8. Integration Testing Summary.....	28

9. Classes Not Covered by Integration Testing.....	28
JavaFX View Classes.....	28
LoginController.....	28
UserManagementController.....	29
Abstract, Utility, and Trivial Classes.....	29
9.1 Summary.....	29
3. System Testing.....	30
1. Purpose and Approach.....	30
2. Test Setup, Tools, and Isolation Strategy.....	30
3. System-Tested Classes and Methods (Coverage List).....	31
3.1 Authentication and Session Management (System Level).....	31
3.2 User Administration (Administrator End-to-End).....	32
3.3 Inventory Management (CRUD and Role Control).....	33
3.4 Billing and Sales Workflow (System Level).....	33
4. Tables for System Test Cases.....	34
LoginFrontendSystemTest.....	34
AuthFlowFrontendSystemTest.....	35
BillingFrontendSystemTest.....	37
ManagerFlowFrontendSystemTest.....	38
UserAdminFrontendSystemTest.....	39
5. Classes Not Covered by System Testing.....	40
JavaFX View and Layout Classes.....	40
Controller Helper and Utility Classes.....	41
Data and Model-Only Classes.....	41
Internal UI Dialog Components.....	41
Summary.....	42

1. Unit Testing

Antea Koxherri was responsible for implementing and documenting **Unit Testing**.

1. Unit Testing Scope and Approach

Unit tests were designed to validate **core business logic** in isolation, avoiding UI (JavaFX) behavior and filesystem operations. Tests were implemented using JUnit 5 and executed successfully in Eclipse.

Where the code involved file persistence (e.g., saving/loading **.dat** files), those methods were excluded from unit testing and documented for integration/system testing instead.

2. Unit-Tested Classes and Methods (Coverage List)

The following classes/methods were selected as unit-testable and were implemented by **Antea Koxherri**:

- **BillingController**

- 1.createNewBill()
2. addItemAtBill(Item, int)
3. removeItemFromBill(SaleItem)
4. getBillTotal()

- **Category**

1. addItem(Item)

2. removeItem(Item)

3. getItems()

4. toString()

- **Supplier**

1.addCategory(Category)

2. addProduct(Item)

3. removeProduct(Item)

4. getProducts()

- **Cashier**

1.addBill(Bill)

2. clearBills()

3. login(String, String)

4. changePassword(String, String)

- **Manager**

1.manageSuppliers(Supplier, String)

2. viewSectorStats(String)

3. changePassword(String, String)

- **Administrator**

1.login(String, String)

2. changePassword(String, String)

3. viewSystemStats()

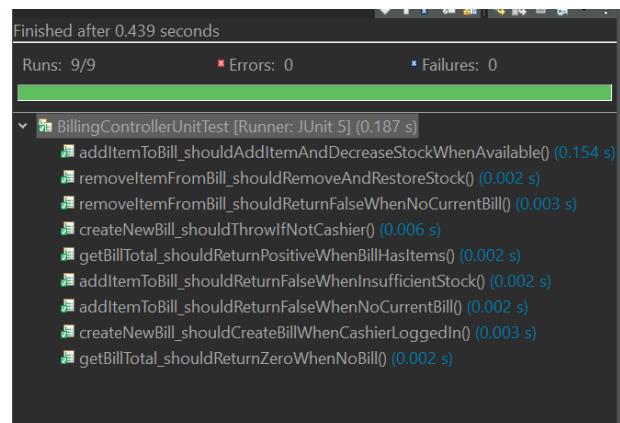
- **SessionState**

1. startSession(User)
2. endSession()
3. setCurrentSection(String)
4. role checks: isCashier(), isManager(), isAdministrator()
5. default state behavior

3. Tables for Test Cases

3.1 BillingController

This unit test set verifies the **core billing logic** that can be tested without external dependencies. The tests focus on bill creation permissions, adding/removing items, stock updates, and safe total calculation.

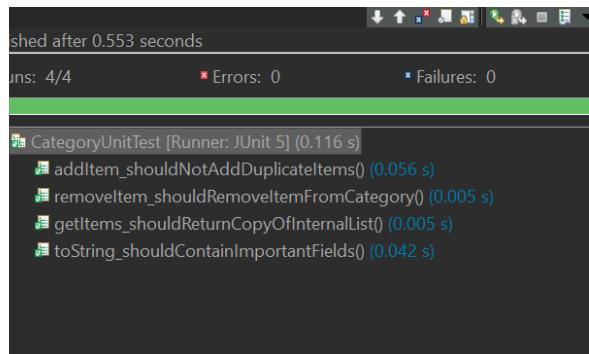


Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type

UT-BC-01	createNewBill	No cashier logged in	IllegalStateException thrown	PASS	Only cashiers can create bills	Unit
UT-BC-02	createNewBill	Cashier logged in	New bill created	PASS	Valid bill creation	Unit
UT-BC-03	addItemToBill	No active bill	Returns false	PASS	Guard condition	Unit
UT-BC-04	addItemToBill	Insufficient stock	Returns false, stock unchanged	PASS	Prevent invalid sale	Unit
UT-BC-05	addItemToBill	Stock available	Returns true, stock decreases	PASS	Correct stock update	Unit
UT-BC-06	removeItemFromBill	No active bill	Returns false	PASS	Guard condition	Unit
UT-BC-07	removeItemFromBill	Valid sale item	Stock restored, item removed	PASS	Reverse stock logic	Unit
UT-BC-08	getBillTotal	No bill	Returns 0.0	PASS	Safe default behavior	Unit
UT-BC-09	getBillTotal	Bill with items	Total > 0	PASS	Correct total calculation	Unit

3.2 Category

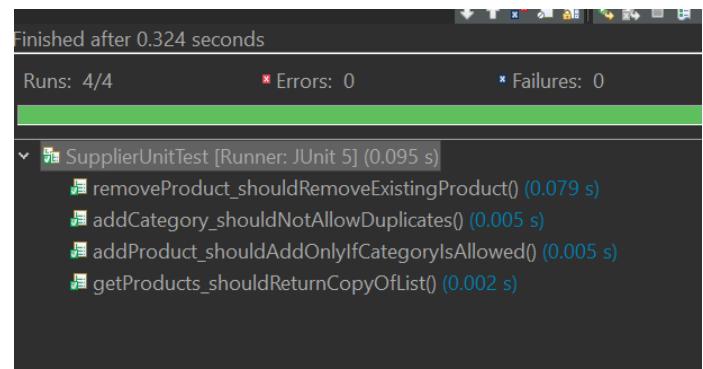
These unit tests validate Category's internal collection management and encapsulation rules. The tests check duplicate prevention, item removal behavior, defensive copying of the items list, and correct string representation output.



Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-CAT-01	addItem	Same item added twice	Only one instance stored	PASS	Prevent duplicates	Unit
UT-CAT-02	removeItem	Existing item	Item removed	PASS	Removal correctness	Unit
UT-CAT-03	getItems	Modify returned list	Internal list unchanged	PASS	Encapsulation	Unit
UT-CAT-04	toString	Standard category	Contains key fields	PASS	Correct string output	Unit

3.3 Supplier

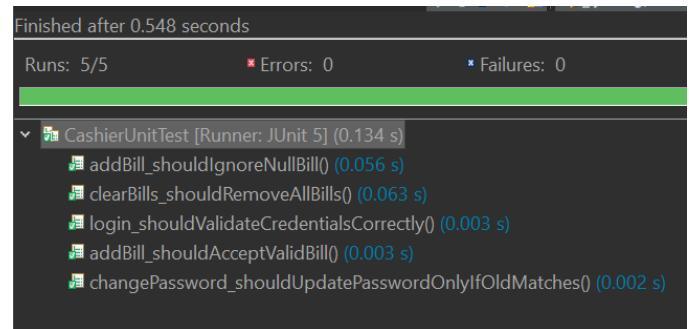
These tests verify Supplier's business rules for managing categories and products, including duplicate category prevention, product admission rules (product must belong to an allowed category), product removal, and returning a defensive copy of the product list.



Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-SUP-01	addCategory	Duplicate category	Category added once	PASS	Prevent duplicates	Unit
UT-SUP-02	addProduct	Allowed vs not allowed category	Only allowed product added	PASS	Business rule validation	Unit
UT-SUP-03	removeProduct	Existing product	Product removed	PASS	Removal correctness	Unit
UT-SUP-04	getProducts	Modify returned list	Internal list unchanged	PASS	Defensive copy	Unit

3.4 Cashier

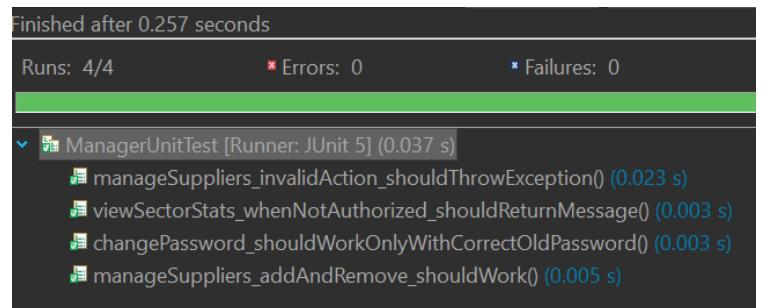
The Cashier unit tests focus on bill list management and authentication/password behavior. The tests validate safe handling of null bills, clearing stored bills, correct login validation, and correct password change behavior.



Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-CASH-01	addBill	Valid bill	Bill accepted	PASS	Bill handling	Unit
UT-CASH-02	addBill	Null bill	Bill ignored	PASS	Null safety	Unit
UT-CASH-03	clearBills	Bills exist	All bills removed	PASS	Reset functionality	Unit
UT-CASH-04	login	Correct / wrong credentials	True only for correct	PASS	Authentication	Unit
UT-CASH-05	changePass word	Wrong then correct old password	Password updated only when valid	PASS	Security logic	Unit

3.5 Manager

These unit tests verify Manager's supplier management workflow, error handling for invalid actions, sector access messaging, and password change logic. Tests confirm correct list behavior (add/remove), proper exception behavior, and correct authorization responses.

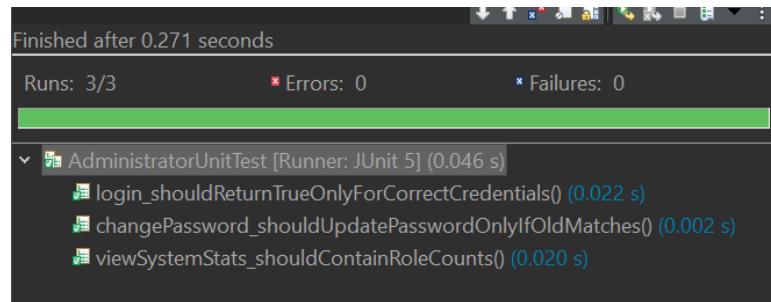


Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-MGR-01	manageSuppliers	Add and remove supplier	Supplier list updated	PASS	Supplier management	Unit
UT-MGR-02	manageSuppliers	Invalid action	Exception thrown	PASS	Input validation	Unit
UT-MGR-03	viewSectorStats	Unauthorized sector	“Not authorized” message	PASS	Access control	Unit

UT-MGR-04	changePass word	Wrong then correct old password	Password updated correctly	PASS	Security logic	Unit
-----------	-----------------	---------------------------------	----------------------------	------	----------------	------

3.6 Administrator

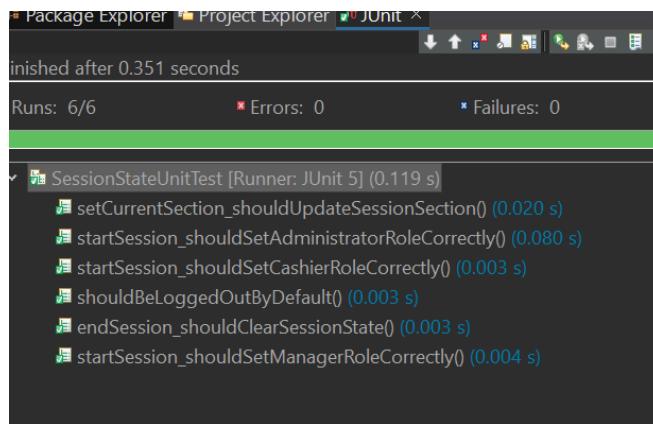
Administrator tests validate authentication, password change correctness, and system statistics generation. The statistics test ensures output includes role-count information after adding different user roles.



Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-ADM-01	login	Correct / wrong credentials	True only for correct	PASS	Authentication	Unit
UT-ADM-02	changePass word	Wrong then correct old password	Password updated	PASS	Security logic	Unit
UT-ADM-03	viewSystemStats	Users added	Role counts shown	PASS	System reporting	Unit

3.7 SessionState

These tests verify session lifecycle state, role detection for different users, section navigation state, and cleanup behavior on logout.



Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
UT-SES-01	default state	No session	Not logged in	PASS	Default behavior	Unit
UT-SES-02	startSession	Cashier user	isCashier = true	PASS	Role detection	Unit
UT-SES-03	startSession	Manager user	isManager = true	PASS	Role detection	Unit
UT-SES-04	startSession	Administrator user	isAdmin = true	PASS	Role detection	Unit
UT-SES-05	setCurrentSection	Section set	Section updated	PASS	State update	Unit
UT-SES-06	endSession	Active session	Session cleared	PASS	Cleanup logic	Unit

4. Methods Not Covered by Unit Testing

BillingController

Some methods of BillingController, such as `finalizeBill()` and `getDailyBills()`, were not tested using unit tests because they depend on reading and writing data to files. These methods use file system operations to save bills or load daily sales, which cannot be easily isolated in unit testing. For this reason, they were excluded from unit tests and are more suitable for integration testing.

InventoryController

The InventoryController methods were not covered by unit testing because they interact directly with persistent storage. Methods such as `addItem()`, `addCategory()`, `addSupplier()`, `updateItemStock()`, `deleteItem()`, and `deleteCategory()` all require reading from or writing to data files. Since unit testing should avoid external dependencies like file systems, these methods were excluded and documented for integration testing.

LoginController

Methods in LoginController, including `login()`, `logout()`, and `changePassword()`, were not unit tested because they depend on both stored user data and the global `SessionState`. These methods coordinate authentication logic across multiple components, making them difficult to test in isolation. Therefore, they were considered integration-level functionality.

UserManagementController

The UserManagementController class was excluded from unit testing because its methods perform complex operations that involve file access and permission checks. Methods such as addUser(), updateUser(), deleteUser(), editUser(), and resetPassword() all modify stored user data and require persistent storage. Due to these dependencies, they were not suitable for unit testing.

ReportController

All methods in ReportController, including generateSalesReport(), generateInventoryReport(), generateLowStockReport(), generateProfitReport(), and exportDataToCSV(), were not unit tested. These methods depend on historical data, file exports, and report formatting, which makes them unsuitable for isolated unit testing. They are better evaluated through integration or system testing.

FileHandler

The FileHandler class was not unit tested because all of its methods perform direct file system operations. Methods such as saveListToFile(), loadListFromFile(), and exportBill() require actual file access, which goes against the principles of unit testing. These methods were therefore excluded from unit tests.

JavaFX View Classes

All JavaFX view classes, such as AdminDashboard, ManagerDashboard, CashierDashboard, and other UI-related classes, were excluded from unit testing. These classes depend on the JavaFX runtime and user interaction, making them unsuitable for unit tests. UI behavior is more appropriately tested using system or manual testing.

Abstract and Trivial Methods

Some methods were not unit tested because they do not contain meaningful business logic. This includes constructors, simple getters and setters, and abstract class methods such as those in the base User class. Since these methods only return values or initialize objects, unit testing them was considered unnecessary.

5. Integration Testing

Jagoda Andrea was responsible for implementing and documenting **Integration Testing**.

5. Integration Testing Scope and Approach

Integration tests were designed to verify the correct interaction between multiple system components, ensuring that different modules work together as expected.

Unlike unit testing, integration testing focuses on validating **end-to-end workflows, data persistence, role-based access control, and system behavior across controllers, models, session management, and file handling**.

The integration tests were implemented using **JUnit 5** and executed in **Eclipse**.

Special attention was given to scenarios involving file persistence (.dat files), session state management, and business workflows such as inventory management, billing operations, and reporting functionalities.

These tests ensure that the system behaves correctly under real usage conditions and that all core modules are properly integrated.

6. Integration Tests – Grouping and Coverage

The implemented integration tests were grouped based on the system components and interactions they validate.

This grouping improves clarity and demonstrates structured test coverage across the system.

6.1 Group 1: File Persistence Integration Tests

These tests verify correct interaction between domain models and the file handling mechanism, ensuring data is safely stored and retrieved.

- 1. UserFileAuthenticationIntegrationTest**

Validates persistence and retrieval of user data using file storage.

- 2. SupplierFileIntegrationTest**

Ensures suppliers can be correctly saved to and loaded from persistent storage.

- 3. InventoryItemFileIntegrationTest**

Tests saving and reading inventory items together with related category and supplier data.

- 4. InventoryFileOverwriteIntegrationTest**

Verifies that existing inventory files are correctly overwritten when new data is saved.

- 5. ReadNonExistingFileIntegrationTest**

Ensures the system handles missing files gracefully without runtime errors.

6.2 Group 2: Inventory and Session Integration Tests

These tests validate role-based access control and inventory management workflows.

- 6. InventorySessionState**

Verifies that inventory actions are allowed or restricted based on the logged-in user role.

7. **InventoryCategoryIntegrationTest**

Tests the complete inventory workflow, including category creation, item addition, and role authorization.

6.3 Group 3: Billing Integration Tests

These tests focus on billing authorization and business logic integration.

8. **BillingRoleIntegrationTest**

Verifies which user roles are permitted to create billing operations.

9. **BillingFlowIntegrationTest**

Tests the complete billing process, including bill creation, item addition, stock updates, and total calculation.

6.4 Group 4: Reporting Integration Tests

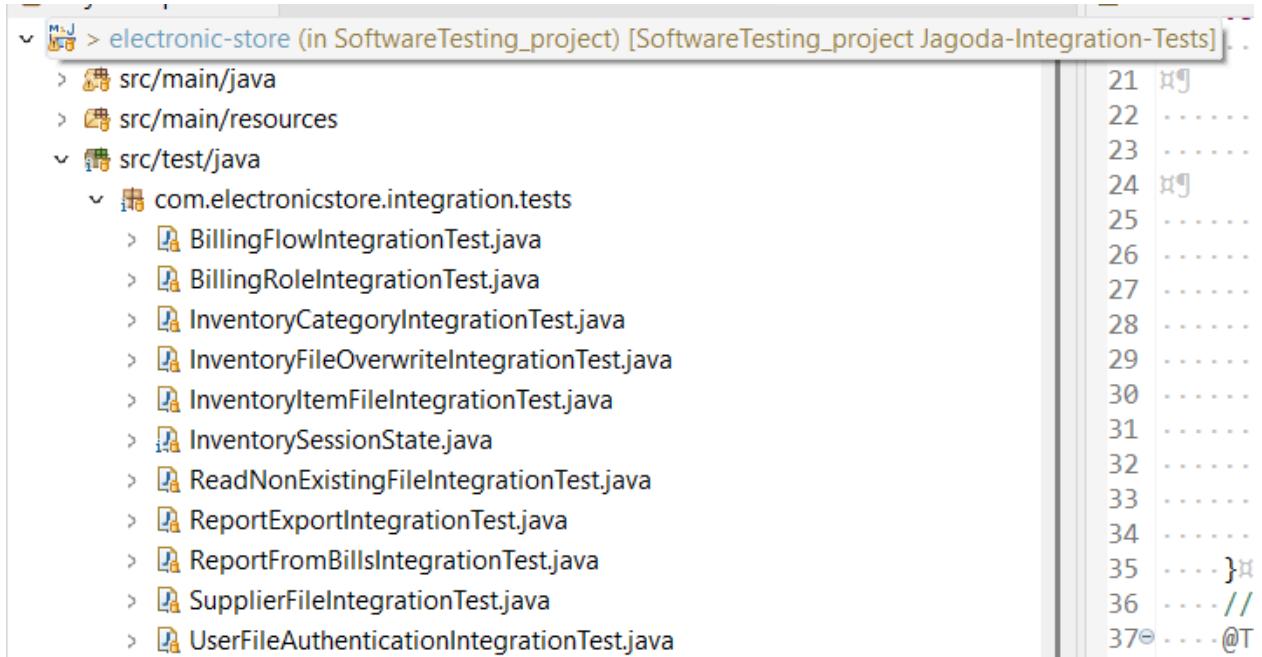
These tests validate reporting data handling and export functionality.

10. **ReportFromBillsIntegrationTest**

Ensures billing data can be persisted and retrieved as input for reporting.

11. **ReportExportIntegrationTest**

Verifies that generated reports can be successfully exported to a text file.



7. *Integration Testing – Method-Level Explanation*

7.1 UserFileAuthenticationIntegrationTest

This integration test verifies the correct interaction between the user domain model and the file persistence mechanism.

The test ensures that user objects can be safely stored in a data file and later retrieved without data corruption or loss.

By validating both saving and reading operations, this test confirms that authentication-related user data remains consistent across system executions.

Finished after 0.31 seconds

Runs: 2/2 Errors: 0 Failures: 0

>UserFileAuthenticationIntegrationTest [Runner: JUnit 5] (0.000 s)

readUsers_whenFileDoesNotExist_returnsEmptyList() (0.000 s)

saveAndReadUsers_successfully() (0.000 s)

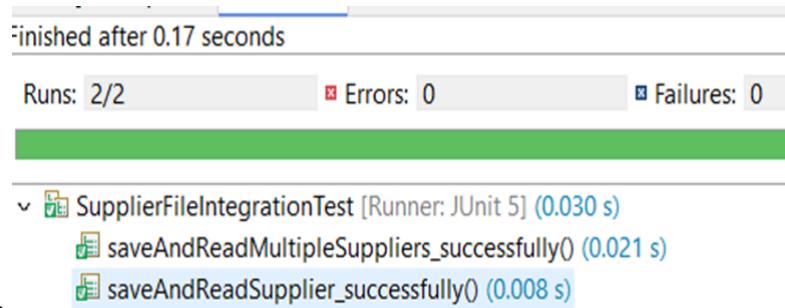
Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
IT-USR-01	saveAndReadUsers_successfully	Save and read a user from file	User data is correctly saved and loaded	PASS	Verify user persistence via file system	Integration
IT-USR-02	readUsers_whenFileDoesNotExist_returnsEmptyList	Read users from missing file	Empty list returned	PASS	Ensure safe handling of missing user files	Integration

7.2 SupplierFileIntegrationTest

This test validates the integration between supplier entities and the file handling utility.

It ensures that supplier information can be persisted and reloaded correctly, both for single and multiple supplier entries.

The test confirms that the system correctly manages collections of supplier objects during file-based operations.



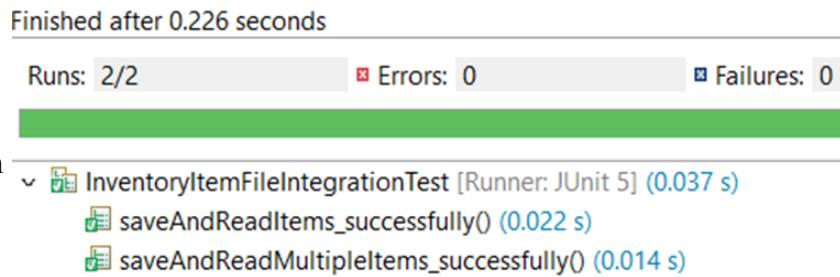
Test-ID	Method	Scenario/Input	Expected Output	Result	Reason for Test	Test Type
IT-SUP-01	<code>saveAndReadSupplier_successfully</code>	Save and read one supplier	Supplier loaded correctly	PASS	Verify supplier file persistence	Integration
IT-SUP-02	<code>saveAndReadMultipleSuppliers_successfully</code>	Save and read multiple supplier	All suppliers loaded correctly	PASS	Validate multiple supplier storage	Integration

7.3 InventoryItemFileIntegrationTest

This integration test verifies the correct persistence of inventory items together with their associated categories and suppliers.

It ensures that complex object relationships are correctly maintained when items are saved to and retrieved from storage.

This test is essential for validating data consistency within the inventory subsystem.



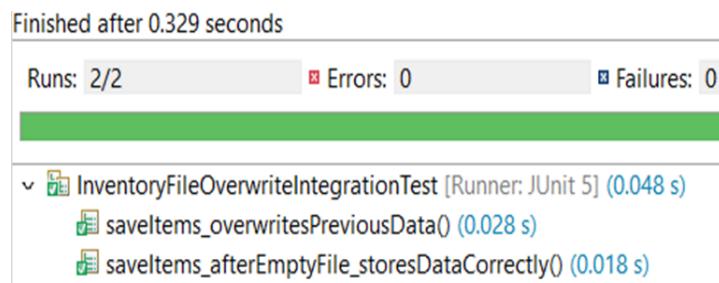
Test ID	Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
IT-INV-ITEM-01	<code>saveAndReadItems_successfully</code>	Save and read one inventory item	Item loaded correctly	PASS	Verify inventory item persistence	Integration
IT-INV-ITEM-02	<code>saveAndReadMultipleItems_successfully</code>	Save and read multiple items	All items loaded correctly	PASS	Validate inventory batch storage	Integration

7.4 InventoryFileOverwriteIntegrationTest

This test focuses on validating correct overwrite behavior when inventory data is saved multiple times to the same file.

It ensures that outdated data is replaced by the most recent information and that no residual data remains after updates.

This behavior is critical for maintaining accurate inventory records.



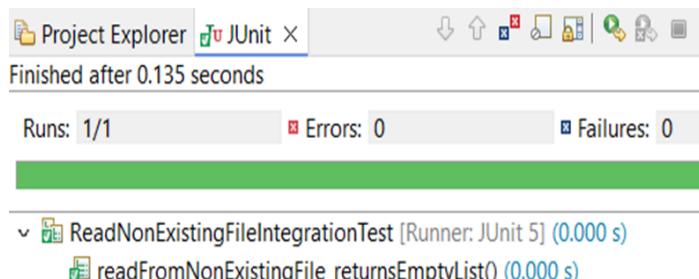
Test-ID	Method	Scenario/Input	Expected Output	Result	Reason for Test	Test Type
IT-INV-OVR-01	saveItems_overwritesPreviousData	Overwrite existing inventory file	Old data replaced	PASS	Verify overwrite behavior	Integration
IT-INV-OVR-02	saveItems_afterEmptyFile_storesDataCorrectly	Save after empty file	Data stored correctly	PASS	Ensure correctly handling of empty files	Integration

7.5 ReadNonExistingFileIntegrationTest

This test verifies the system's behavior when attempting to read data from a file that does not exist.

Instead of causing runtime failures, the system is expected to handle this scenario gracefully by returning an empty data set.

The test confirms the robustness and fault tolerance of the file handling mechanism.



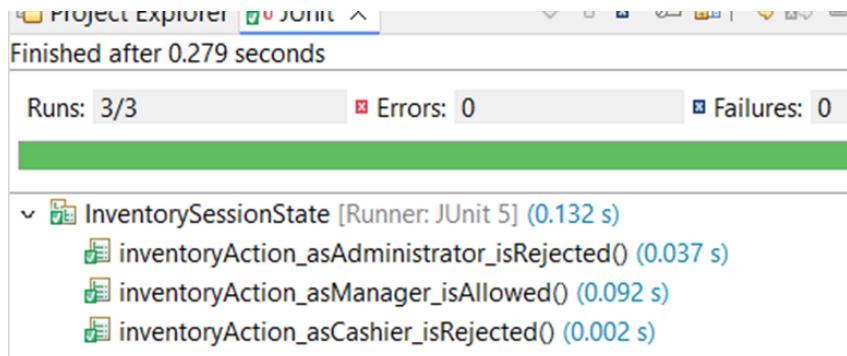
<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-FILE-01	<code>readFromNonExistingFile_returnsEmptyList</code>	Read non-existing file	Empty list returned	PASS	Ensure system does not crash on missing files	Integration

7.6 InventorySessionState

This integration test validates role-based access control within the inventory module.

It ensures that only authorized users, specifically those with the Manager role, are allowed to perform inventory modification actions.

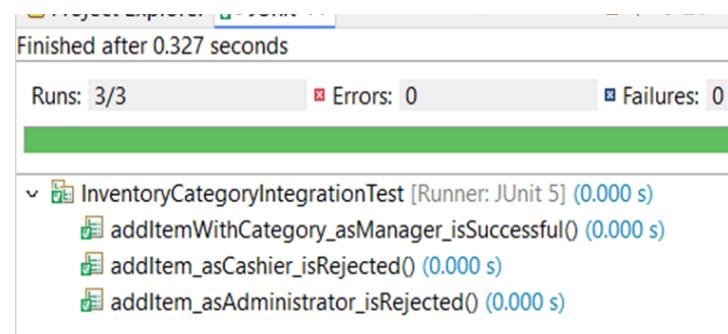
The test confirms proper integration between session management, user roles, and inventory controller logic.



<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-INV-SES-01	<code>inventoryAction_asManager_isAllowed</code>	Manager session	Action allowed	PASS	Verify session-based access	Integration
IT-INV-SES-02	<code>inventoryAction_asCashier_isRejected</code>	Cashier session	Action rejected	PASS	Enforce role-based control	Integration
IT-INV-SES-03	<code>inventoryAction_asAdministrator_isRejected</code>	Administrator session	Action rejected	PASS	Enforce role-based control	Integration

7.7 InventoryCategoryIntegrationTest

This test verifies the complete inventory management workflow, including category creation and item addition.



It ensures that categories created within the system can be retrieved and used correctly when adding inventory items.

The test also confirms that role-based permissions are enforced during these operations.

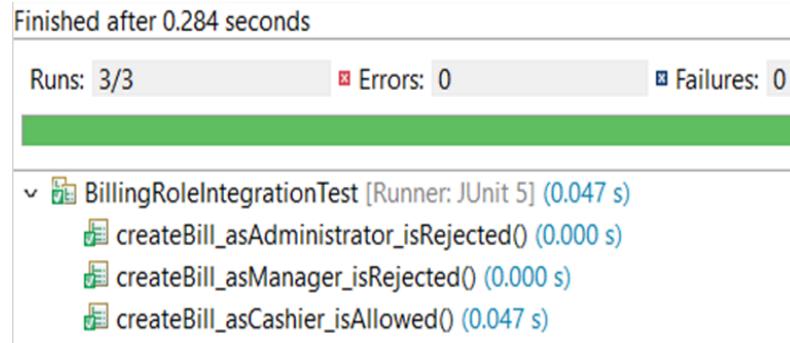
<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-INV-CAT-01	addItemWithCategory_asManager_isSuccessful	Manager adds item	Item added successfully	PASS	Verify manager inventory permissions	Integration
IT-INV-CAT-02	addItem_asCashier_isRejected	Cashier adds item	Operation rejected	PASS	Enforce cashier restrictions	Integration
IT-INV-CAT-03	addItem_asAdministrator_isRejected	Administrator adds item	Operation rejected	PASS	Enforce administrator restrictions	Integration

7.8 BillingRoleIntegrationTest

This integration test focuses on validating authorization rules within the billing subsystem.

It ensures that only users with the Cashier role are permitted to create billing operations, while other roles are restricted.

The test confirms correct interaction between session state management and billing controller authorization logic.

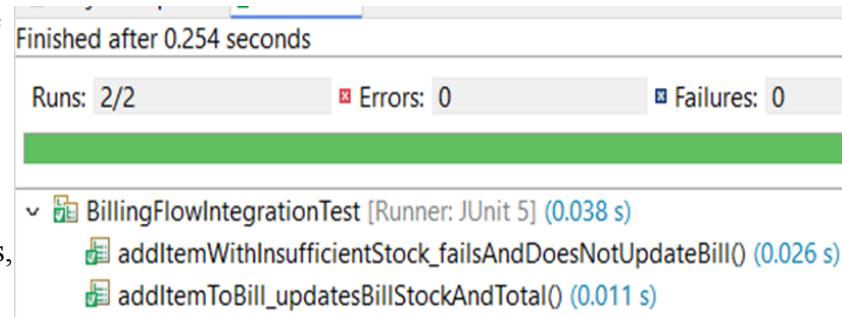


<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-BILL-ROLE-01	<code>createBill_asCashier_isAllowed</code>	Cashier creates bill	Bill created	PASS	Verify billing workflow	Integration
IT-BILL-ROLE-02	<code>createBill_asManager_isRejected</code>	Manager creates bill	Exception thrown	PASS	Validate stock availability	Integration
IT-BILL-ROLE-03	<code>createBill_asAdmin_isRejected</code>	Administrator creates bill	Exception thrown	PASS	Validate stock availability	Integration

7.9 BillingFlowIntegrationTest

This test validates the complete billing workflow as a single integrated process.

It verifies bill creation, item addition, stock quantity updates, and total price calculation.



By testing the entire flow end-to-end, this test ensures that billing operations behave correctly under realistic usage conditions.

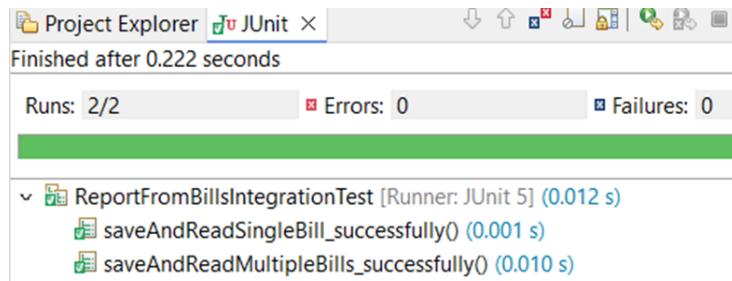
<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-BILL-FLOW-01	<code>addItemToBill_updatesBillStockAndTotal</code>	Add item with sufficient stock	Stock and total updated	PASS	Verify billing workflow	Integration
IT-BILL-FLOW-02	<code>addItemWithInsufficientStock_failsAndDoesNotUpdateBill</code>	Add Item with insufficient stock	Operation fails safely	PASS	Validate stock availability	Integration

7.10 ReportFromBillsIntegrationTest

This integration test verifies that billing data can be persisted and retrieved correctly for reporting purposes.

It ensures that bills stored in persistent storage remain available and accurate when used as input for report generation.

The test confirms the integrity of reporting data sources.



Test-ID	Method	Scenario/Input	Expected Output	Result	Reason for Test	Test Type
IT-REP-BILL-01	saveAndReadSingleBill_successfully	Save and read single bill	Bill loaded correctly	PASS	Ensure bill data is stored for reporting	Integration
IT-REP-BILL-02	saveAndReadMultipleBills_successfully	Save and read multiple bills.	All bills loaded correctly	PASS	Validate report consistency from bills	Integration

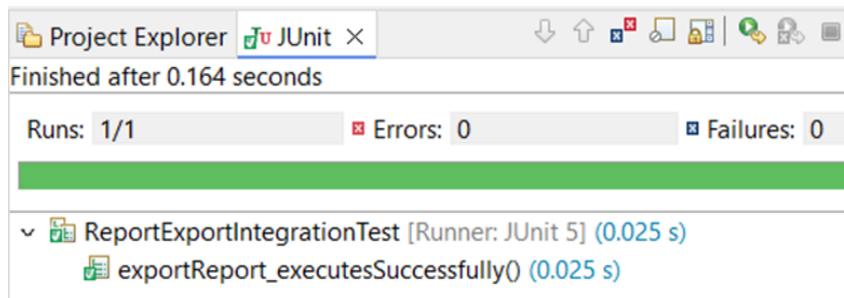
7.11 ReportExportIntegrationTest

This test validates the report export functionality by ensuring that generated report content can be successfully written to a text file.

It confirms correct interaction

between the reporting logic and file output mechanisms.

This test ensures that final reports can be produced and shared as expected.



<i>Test-ID</i>	<i>Method</i>	<i>Scenario/Input</i>	<i>Expected Output</i>	<i>Result</i>	<i>Reason for Test</i>	<i>Test Type</i>
IT-REP-EXP-01	exportReport_executesSuccessfully	Export report to file	Export operation succeeds	PASS	Verify report export functionality	Integration

8. Integration Testing Summary

Together, these integration tests provide comprehensive coverage of the system's core interactions.

They validate data persistence, role-based access control, business workflows, and reporting functionality, ensuring that all major components collaborate correctly within the integrated system.

9. Classes Not Covered by Integration Testing

Although integration testing validates interactions between multiple system components, certain classes were intentionally excluded from integration testing.

These exclusions were based on testing scope, feasibility, and best practices in software testing.

JavaFX View Classes

All JavaFX view classes were excluded from integration testing.

These classes are responsible for rendering the user interface and handling user interactions, which fall outside the scope of backend integration testing.

UI-related behavior is more appropriately validated through manual testing or system-level acceptance testing.

LoginController

The LoginController class was not covered by integration testing.

This controller primarily manages user input validation and UI-driven authentication logic.

Since integration testing focused on backend component interaction, login behavior was validated indirectly through session-based tests rather than direct controller testing.

UserManagementController

The UserManagementController was not covered by integration testing.

This controller is closely tied to administrative UI workflows and user interface interactions.

User-related persistence and role-based access control were validated through other integration tests, making direct testing of this controller unnecessary.

Abstract, Utility, and Trivial Classes

Abstract base classes, trivial utility classes, and data-only classes (such as getters, setters, and simple data holders) were excluded from integration testing.

These classes do not contain meaningful integration logic and therefore do not provide additional value when tested at the integration level.

9.1 Summary

The integration testing strategy focused on validating meaningful backend interactions involving controllers, models, session management, and persistence mechanisms.

Classes excluded from integration testing were either UI-focused or trivial in nature and were intentionally left out to maintain a clear and efficient testing scope.

3. System Testing

Flavia Koço was responsible for implementing and documenting **System Testing**.

1. Purpose and Approach

System testing was carried out to verify that the Electronics Store application functions correctly as a **complete, integrated system** when used by real users. The primary goal was to validate full end-to-end workflows, such as registration, login, inventory management, billing, reporting, and user administration, rather than testing individual components in isolation.

To achieve this, system tests simulate realistic user interaction with the running application, including typing into forms, selecting options, clicking buttons and menus, opening dialogs, and confirming actions. These interactions validate not only the correctness of the user interface but also that each UI action correctly triggers backend logic, enforces business rules, updates session state, and persists data to files.

The system tests were implemented using **JUnit 5** as the test framework and **TestFX** to automate JavaFX user interface interaction. This allowed the application to be tested in the same way an end user would operate it, while still enabling automated verification of backend state changes.

2. Test Setup, Tools, and Isolation Strategy

System testing was implemented using **JUnit 5** in combination with **TestFX**, which enables automated interaction with JavaFX applications. All system tests launch the real application (`com.electronicstore.App`) and interact with live UI screens and dialogs.

All system test classes extend `BaseFrontendSystemTest`, which provides a consistent setup and guarantees isolation between test cases by:

- Launching the JavaFX application once per test run
- Forcing each test to start from the login screen
- Waiting for JavaFX events to complete to avoid timing issues caused by asynchronous UI updates

Additional isolation was achieved by:

- Deleting or resetting saved data files (users.dat, items.dat, categories.dat, suppliers.dat, bills.dat) where required
- Explicitly ending any active session using *SessionState.endSession()*
- Seeding required users or inventory through backend controllers when necessary

This strategy ensures that each system test executes in a clean and predictable environment while still validating real persistence and end-to-end system behavior.

3. System-Tested Classes and Methods (Coverage List)

The following system-level behaviors were validated through system tests that interact with the JavaFX user interface and verify backend outcomes.

3.1 Authentication and Session Management (System Level)

System Tests Involved

- AuthFlowFrontendSystemTest
- LoginFrontendSystemTest

Classes Exercised

- LoginController
- UserManagementController
- Administrator
- Manager
- Cashier
- SessionState

System-Tested Behaviors

- User registration through the UI for Administrator, Manager, and Cashier roles

- Login and logout flows through the UI
- Session initialization, role assignment, and session termination
- Role-based dashboard and menu visibility after login
- Handling of empty login submissions
- Handling of invalid login credentials

3.2 User Administration (Administrator End-to-End)

System Tests Involved

- UserAdminFrontendSystemTest

Classes Exercised

- UserManagementController
- Administrator
- User
- SessionState

System-Tested Behaviors

- Administrator login and dashboard access
- Creating users through UI dialogs
- Editing user details (name, email, phone, role, sector) via UI
- Resetting user passwords through the UI
- Verifying login with the default reset password
- Deleting users through the system interface
- Persistence of user changes across sessions

3.3 Inventory Management (CRUD and Role Control)

System Tests Involved

- ManagerFlowFrontendSystemTest

Classes Exercised

- InventoryController
- Item
- Category
- Supplier
- Manager
- Cashier
- SessionState

System-Tested Behaviors

- Adding categories and suppliers through manager dialogs
- Adding inventory items through the UI
- Viewing inventory data in tables
- Enforcing role-based restrictions (cashiers blocked, managers allowed)
- Persistence of inventory data after UI operations

3.4 Billing and Sales Workflow (System Level)

System Tests Involved

- BillingFrontendSystemTest

Classes Exercised

- BillingController
- Bill
- SaleItem

- InventoryController
- Cashier
- Manager

System-Tested Behaviors

- Creating new bills through the cashier UI
- Adding items to a bill with quantity selection
- Preventing overselling through UI validation
- Handling empty bill finalization attempts
- Finalizing bills successfully
- Updating and persisting stock quantities after bill finalization
- Displaying generated bill details to the user

4. *Tables for System Test Cases*

LoginFrontendSystemTest

This system test verifies the correctness of the authentication entry point of the application. It ensures that the login screen properly handles invalid input cases such as empty credentials and incorrect usernames or passwords, and that successful registration followed by login correctly transitions the user into the main application interface.

The screenshot shows a terminal window displaying the results of a JUnit test run. At the top, it says "Finished after 23.927 seconds". Below that, it shows "Runs: 3/3", "Errors: 0", and "Failures: 0". A green bar indicates success. The main area lists the test class "LoginFrontendSystemTest [Runner: JUnit 5] (23.426 s)" with three test methods: "register_then_login() (18.116 s)", "wrong_login_shows_error() (3.632 s)", and "empty_login_shows_error() (1.676 s)". All three tests are marked with a green checkmark icon.

```

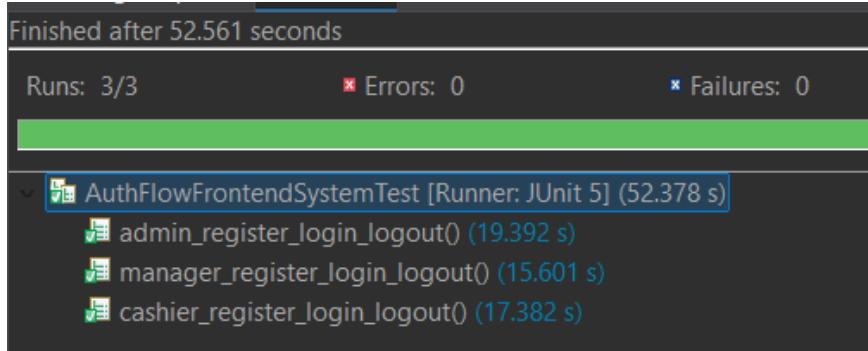
Finished after 23.927 seconds
Runs: 3/3          ✘ Errors: 0          ✘ Failures: 0
[green bar]
LoginFrontendSystemTest [Runner: JUnit 5] (23.426 s)
  ✓ register_then_login() (18.116 s)
  ✓ wrong_login_shows_error() (3.632 s)
  ✓ empty_login_shows_error() (1.676 s)

```

Test ID	Test Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
ST-UI-LOGIN-01	empty_login_shows_error()	Click Login with empty fields	Login error dialog shown; remains on login screen	PASS	Validate empty input handling	System
ST-UI-LOGIN-02	wrong_login_shows_error()	Enter invalid credentials	Login error dialog shown; remains on login screen	PASS	Validate incorrect credential handling	System
ST-UI-LOGIN-03	register_then_login()	Register cashier and log in	Login succeeds; main menu visible	PASS	Validate registration and login flow	System

AuthFlowFrontendSystemTest

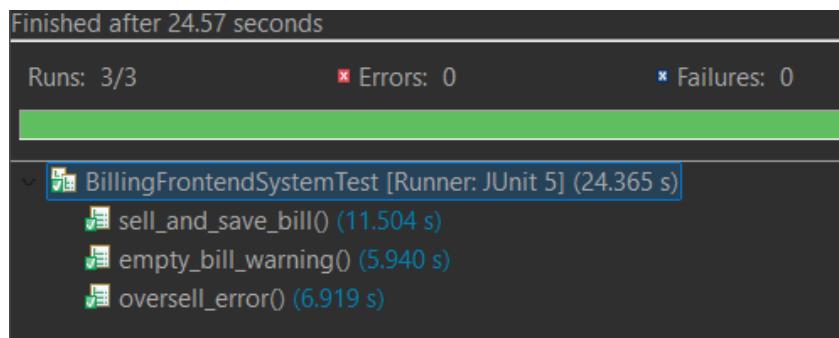
This system test validates the full authentication workflow for all user roles through the JavaFX interface. It verifies that users can register, log in, and log out successfully, that the system correctly identifies their role, and that the appropriate menus and dashboard elements are displayed based on the logged-in user's permissions.



Test ID	Test Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
ST-UI-AUTH-0 1	admin_register_login_logout()	Register and log in as Administrator	Admin menus visible; logout returns to login	PASS	Validate administrator UI flow	System
ST-UI-AUTH-0 2	manager_register_login_logout()	Register and log in as Manager	Manager dashboard buttons visible;	PASS	Validate manager UI flow	System
ST-UI-AUTH-0 3	cashier_register_login_logout()	Register and log in as Cashier	Cashier dashboard/billing buttons visible	PASS	Validate cashier UI flow	System

BillingFrontendSystemTest

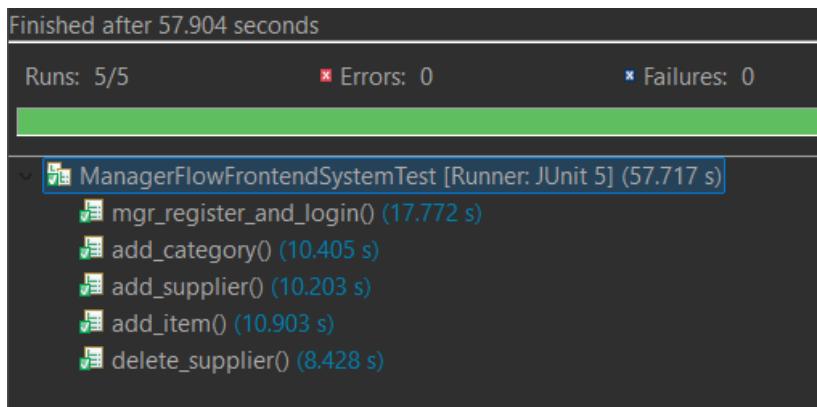
This system test verifies the cashier billing process as it is performed through the graphical interface. It ensures that bill creation, item selection, quantity handling, and bill finalization behave correctly, and that errors such as overselling or attempting to finalize an empty bill are properly handled by the UI while correctly updating backend data.



Test ID	Test Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
ST-Uİ-BILL-01	sell_and_save_bill()	Create bill, add item, finalize	Success dialog; receipt shown; stock updated	PASS	Validate complete billing workflow	System
ST-Uİ-BILL-02	empty_bill_warning()	Finalize bill with no items	Warning dialog shown	PASS	Prevent empty bill finalization	System
ST-Uİ-BILL-03	oversell_error()	Add quantity greater than stock	Error dialog shown	PASS	Prevent overselling via UI	System

ManagerFlowFrontendSystemTest

This system test validates the manager's interaction with inventory management features through the JavaFX interface. It confirms that a manager can register, log in, and perform inventory-related operations such as adding categories, suppliers, and items, as well as deleting suppliers, using dialog-based workflows.



Test ID	Test Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
ST-UI-MGR-01	mgr_register_and_login()	Register and log in as Manager	Manager dashboard visible	PASS	Validate manager authentication	System
ST-UI-MGR-02	add_category()	Add category via UI	Category appears in table	PASS	Validate category creation	System
ST-UI-MGR-03	add_supplier()	Add supplier via UI	Supplier appears in table	PASS	Validate supplier creation	System

ST-UI-MGR-04	add_item()	Add item via UI	Item appears in inventory table	PASS	Validate item creation	System
ST-UI-MGR-05	delete_supplier()	Delete supplier via UI	Supplier removed from table	PASS	Validate supplier deletion	System

UserAdminFrontendSystemTest

This system test verifies administrator-level user management through the graphical interface. It ensures that an administrator can add new users, edit existing user details, reset user passwords, and delete users, and that all changes are correctly reflected in the underlying system data.

```
Finished after 45.011 seconds
Runs: 4/4          ✘ Errors: 0      ✘ Failures: 0
UserAdminFrontendSystemTest [Runner: JUnit 5] (44.843 s)
  edit_user() (16.232 s)
  add_user() (14.333 s)
  delete_user() (7.170 s)
  reset_password() (7.102 s)
```

Test ID	Test Method	Scenario / Input	Expected Output	Result	Reason for Test	Test Type
ST-UI-ADMIN-01	reset_password()	Reset user password via UI	Password reset to default value	PASS	Validate password reset	System
ST-UI-ADMIN-02	edit_user()	Edit user details via UI	User details updated	PASS	Validate edit user flow	System
ST-UI-ADMIN-03	delete_user()	Delete user via UI	User removed from system	PASS	Validate delete user flow	System
ST-UI-ADMIN-04	add_user()	Add new user via UI	New user created	PASS	Validate add user flow	System

5. *Classes Not Covered by System Testing*

Although system testing validates complete end-to-end application workflows, not all classes were tested as standalone entities at the system level. These exclusions were intentional and based on testing scope, feasibility, and established software testing best practices.

JavaFX View and Layout Classes

Individual JavaFX view and layout classes were not tested in isolation at the system level. However, they were **validated indirectly through UI-driven system tests implemented using TestFX**.

User interactions such as clicking buttons, opening dialogs, filling in forms, navigating menus, and verifying visible UI elements confirm that these views and layouts function correctly as part of complete system workflows. Since system testing focuses on real user behavior rather than internal UI structure, separate system tests for individual view or layout classes were considered unnecessary.

Controller Helper and Utility Classes

Low-level helper and utility classes that support controller functionality were not directly targeted by system tests. These classes do not represent user-observable system behavior and are therefore not suitable for direct system-level validation.

Their correctness was exercised **indirectly** through higher-level system workflows such as authentication, inventory management, billing, reporting, and user administration, where failures would surface if helper logic were incorrect.

Data and Model-Only Classes

Simple data holder and model classes containing basic fields, constructors, getters, and setters were not directly tested at the system level. These classes were already validated through **unit and integration testing**, and additional system tests would not provide meaningful additional coverage.

At the system level, these models were implicitly validated when full workflows completed successfully and produced correct system outcomes (e.g., persisted users, updated inventory, finalized bills).

Internal UI Dialog Components

Specific UI dialog components (such as add/edit dialogs, confirmation dialogs, and warning dialogs) were not tested as independent system entities. Instead, their correctness was validated implicitly when system workflows involving these dialogs completed successfully and produced the expected backend and UI outcomes.

System tests confirmed that dialogs:

- Open when expected
- Accept valid input
- Block invalid operations
- Trigger correct backend behavior upon confirmation

Summary

The system testing strategy focused on validating **realistic end-to-end user workflows** rather than isolated class behavior. Classes not directly covered by system testing were either:

- Validated indirectly through UI-driven system tests using TestFX
- Already covered by unit or integration testing
- Not suitable for standalone system-level validation

This approach ensured **efficient, meaningful, and non-redundant system test coverage**, while maintaining confidence that the Electronics Store application behaves correctly as a complete system.