

POS-System

SWE-303

Project



Course: **SWE 303 – Software Testing & Quality Assurance**

Accepted by: **Ari Gjerazi, Jurgen Cama**

Group Members:

Ester Shumeli

Abdulaziz Bezan

Eridjon Krrashi

Evisa Nela



Github Repository: <https://github.com/Arkbezo/Software-Testing-Project-Fall-2025>

PROJECT OVERVIEW

Java-based **Point of Sale (POS)** system developed as a case study for software testing

The system manages **users, products, customers, suppliers, and transactions**

Main objective is to ensure **software quality, correctness, and reliability**

Testing activities were performed **strictly according to project guidelines**

The project applies **multiple testing levels**, including:

- Static Testing
- Testing Analysis
- Unit Testing
- Integration Testing
- System Testing



Different **testing techniques** were used, such as:

- Boundary Value Testing
- Equivalence Class & Decision Table Testing
- Code Coverage and MC/DC
- Static Code Analysis (SpotBugs & SonarQube)

Each group member contributed **individually**, with **distinct responsibilities**, ensuring:

- No duplication of work
- Clear accountability
- Full testing coverage of the system

Task Distribution Overview

Ester Shumeli

01

- Group Leader, task division, check/document overall progress
- Testing Analysis
- Integration Testing

Abdulaziz Bezan

02

- Vice Team Leader
- Testing Analysis
- System Testing

Eridjon Krrashi

03

- Static Testing
- Testing Analysis
- Unit Testing

Evisa Nela

04

- Testing Analysis
- Integration Testing

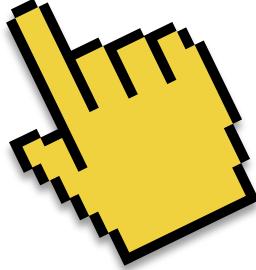


Customer Information			
	Id	Name	Phone No
	1	fawad	12345
	3	fawad iqbal	1234568
	6	saman	77-88-9
	9	sam	9999988
	10	fawad	03149972883
	27	saman	77-88-9
	38	fawad Iqbal	03149972883
	40	wertyq	1234567890
	41	fawad iqbal	1234568133
	43	fawad Iqbal	03149972883
	44	fawad iqbal	1234568133
	45	fawad Iqbal	03149972883
	46	fawad Iqbal	03149972883
	47	fawad Iqbal	03149972883
	48	fawad iqbal	1234568133
	49	fawad Iqbal	03149972883
	50	fawad Iqbal	03149972883
	51	fawad Iqbal	03149972883
	52	fawad Iqbal	03149972883
	53	fawad Iqbal	03149972883
	54	fawad Iqbal	03149972883
	55	wertyq	1234567890
	57	ahmad1	1234567899
	58	fawad iqbal	03149972883
	62	OtherName_IntTest_456e66	0690000002

[Add](#)[Delete](#)[Update](#)

Search by Name:

Static Testing



Static testing evaluates the system without executing the code

Focus areas:

- Code quality
 - Maintainability
 - Potential runtime defects

Tools used:

- SonarQube
 - SpotBugs
 - Manual Code Review

The screenshot shows the SonarQube interface with several tabs open:

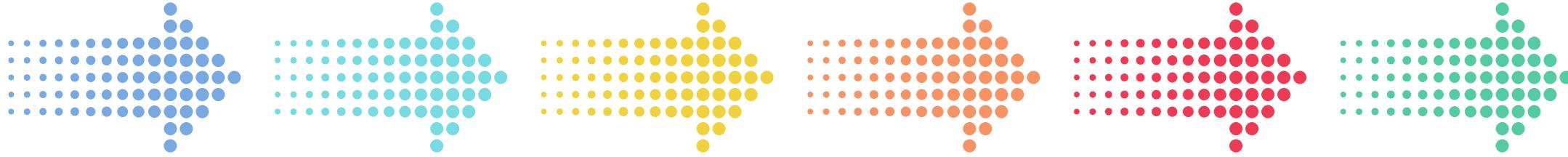
- SonarQube Report**: Shows a summary of 10 items found, including 10 bugs.
- SonarQube Bindings**: Shows 10 items found, including 10 bugs.
- Bug Explorer**: Shows 10 items found, including 10 bugs.
- Bug Info**: Shows detailed information for a specific bug.
- SonarQube Report**: Shows detailed inspection results for a file named `src/main/java/com/sonarqube/SonarQubeConsole.java`. The report highlights several issues related to string literals being duplicated across multiple methods. A specific issue is highlighted with a red circle and the message: "String literals should not be duplicated".

The code snippet from `SonarQubeConsole.java` is as follows:

```
10 11     this.objector = P0Factory.getInstanceForServer();
12 13     this.jdbcHandler = P0Factory.getJdbcHandlerForServer();
14 15     this.jdbcHandler = P0Factory.getJdbcHandlerForServer();
16 17 }
18
19 //overriden
20 public void verifyUser(UserDTO user, Response responseObj) {
21     Connection connection = mySQL.getConnection();
22
23     if (connection == null) {
24         Message message = new Message(DatabaseConnectionIssue.please_contact_customer_services(), MessageType.Exception);
25         responseObj.setMessageObject(message);
26     } else {
27         String query = "SELECT * FROM users WHERE username = ? AND password = ?";
28         PreparedStatement preparedStatement = connection.prepareStatement(query);
29         preparedStatement.setString(1, user.getUsername());
30         preparedStatement.setString(2, user.getPassword());
31
32         if (responseObj.isSuccessful()) {
33             User userObj = user;
34             userObj.verifyUser(resultset, user, responseObj);
35         }
36         mySQL.closeConnection(connection);
37     }
38
39 //overriden
40 public void addAddress(UserDTO userObj, Response responseObj) {
41     Connection connection = mySQL.getConnection();
42
43     if (connection == null) {
44         Message message = new Message(DatabaseConnectionIssue.please_contact_customer_services(), MessageType.Exception);
45         responseObj.setMessageObject(message);
46     } else {
47         Address address = addAddress(connection, responseObj);
48         mySQL.closeConnection(connection);
49     }
50 }
```

The inspection details for the highlighted issue show the following results:

File	Line	Method	Message	Severity
SonarQubeConsole.java	10	private void verifyUser(UserDTO user, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void addAddress(UserDTO userObj, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void addAddress(Address address, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void closeConnection(Connection connection)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void addAddress(Address address, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void closeConnection(Connection connection)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void addAddress(Address address, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void closeConnection(Connection connection)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void addAddress(Address address, Response responseObj)	String literals should not be duplicated	INFO
SonarQubeConsole.java	10	private void closeConnection(Connection connection)	String literals should not be duplicated	INFO



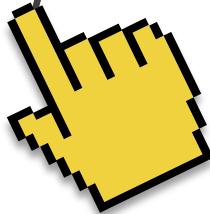
Key Static Testing Findings

- Duplicate string literals (database error messages)
 - Missing null checks on database connections
 - Methods returning null instead of empty collections
 - Use of concrete return types (ArrayList) instead of interfaces (List)
 - DALManager identified as the **most critical class**



Performed by: Eridjon Krrashi

Testing Analysis



Ester Shumeli

Methods Analyzed:

isValidPassword(...)
searchCustomersByName(...)
isValidPhoneNo(...)

Result:

All logical boundaries verified
Independent condition impact proven

Each member performed **individual Testing Analysis** & selected **distinct methods**
Analysis techniques used:

- Boundary Value Testing (BVT)
- Equivalence Class / Decision Table Testing
- Code Coverage & MC/DC



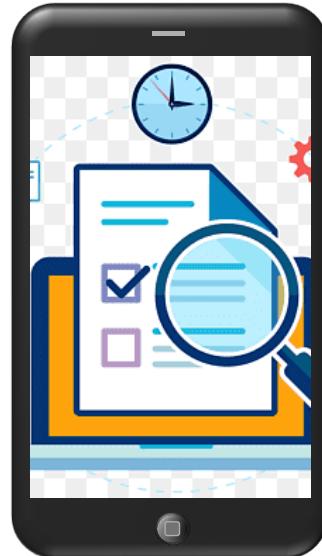
Abdulaziz Bezan

Methods Analyzed:

- calculateTotal()
- addToCartBtnActionPerformed()
- create_invoiceActionPerformed()

Focused on:

- Cart total calculation logic
- User interaction edge cases
- Identification of **missing input validation and error handling gaps**

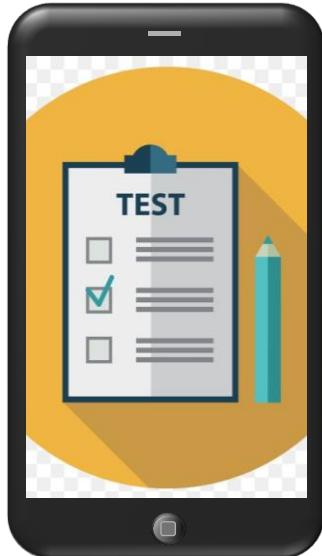


Eridjon Krrashi

Class Analyzed: DALManager

Coverage Achieved:

100% Statement Coverage
100% Branch Coverage
100% Condition Coverage



Evisa Nela

Methods Analyzed:

isSessionExpired()
isSuccessfull()
getErrorMessages()

Result:

Time-based logic verified
Correct message filtering and output behavior

Unit Testing



Performed by: Eridjon Krrashi

Tested **core logical classes**, including:

- DALManager (data access & search operations)
 - ObjectAdder (insert operations & foreign key handling)
 - ObjectModifier (update operations)
 - ObjectRemover (delete operations)
 - ObjectMapper (ResultSet → DTO mapping)
 - CommonValidator (input validation rules)



A collage of various icons representing technology, data, and connectivity. It includes a large blue clock, a tablet displaying binary code, a smartphone showing a line graph, a desktop monitor with binary data, a laptop, a gear, a cloud with a signal, a gear, a mail icon, and a small robot-like figure.

Unit tests covered:

- CRUD operations for all entities
 - Search functionality and data retrieval
 - Boundary values for IDs (-1, 0, valid ID, MAX_INT)
 - Equivalence class testing (valid, invalid, null, empty inputs)
 - Safe handling of null values and SQL exceptions



Results:

Over 60 unit test cases executed
100% pass rate across all tested classes
No NullPointerExceptions or runtime
crashes detected
All critical POS functionality validated in
isolation

Start Integration Test

Execute Database Operations
(Insert / Update / Delete)

Run JUnit Assertions

Assertions Passed Successfully

Foreign Key Constraints Enforced

Database State Consistent

Test Result: PASS

- ✓ All assertions passed successfully
- ✓ Foreign key constraints enforced correctly
- ✓ Database state remained consistent

Integration Testing



: Evisa Nela



Test 1: Product Stock Update & Verification

- Insert product into database
- Retrieve product and verify initial stock
- Update stock quantity and re-verify
- Delete product after test (cleanup)
- Confirms correct interaction between **Product module** and **database layer**

Test 2: Supplier Delete Constraint (Parent–Child)

Insert supplier and linked product

Attempt supplier deletion (blocked by FK constraint)

Delete dependent product

Successfully delete supplier

Validates **referential integrity enforcement**



The screenshot shows an IDE interface with several files open:

- Project tree: Shows modules like `src/main/java`, `src/test/java`, and `src/test/resources`.
- Code editor: The `IntegrationProductStockTest.java` file is active, containing JUnit test cases for updating product stock.
- Code editor: The `IntegrationProductStockTest.java` file is also visible in the background.
- Code editor: The `ApplicationSession.java` file is partially visible.
- Code editor: The `DriverManager.getConnection()` method is shown, connecting to a MySQL database.
- Code editor: The `productStockUpdateAndVerify_shouldWork()` test method is shown, which performs an update and a select query to verify the result.
- Code editor: The `IntegrationProductStockTest` class definition is shown.

```
public class IntegrationProductStockTest {  
    void productStockUpdateAndVerify_shouldWork() throws Exception {  
        PreparedStatement update = conn.prepareStatement(  
            "UPDATE products SET stock_quantity = ? WHERE id = ?");  
        update.setDouble(1, 25);  
        update.setInt(2, productId);  
        update.executeUpdate();  
  
        PreparedStatement select2 = conn.prepareStatement(  
            "SELECT stock_quantity FROM products WHERE id = ?");  
        select2.setInt(1, productId);  
        ResultSet rs2 = select2.executeQuery();  
        assertTrue(rs2.next());  
        assertEquals(expected, rs2.getDouble("stock_quantity"));  
  
        PreparedStatement delete = conn.prepareStatement(  
            "DELETE FROM products WHERE id = ?");  
        delete.setInt(1, productId);  
    }  
}
```

Integration Testing

: Ester Shumeli



Designed and executed multiple integration tests covering core POS subsystems

Tests were implemented using JUnit with both real database integration and stubs, depending on the strategy

Integration strategies applied:

- **Decomposition-Based Integration** (User insertion & authentication)
- **Top-Down Integration** (Checkout process using stubs)
- **Bottom-Up Integration** (Validation logic, DAL workflows)
- **Sandwich (Hybrid) Integration** (Product–Category relationship)
- **Big Bang Integration** (Employee CRUD subsystem)

Key scenarios tested:

- User creation, retrieval, and authentication
- Checkout logic: stock validation, total calculation, sale persistence
- Customer phone number validation across validator layers
- Product–Category and Product–Supplier foreign key enforcement
- Customer and Employee full CRUD workflows (insert → search → update → delete)

```
src/test/java
  └── IntegrationTesting
      ├── CheckoutService.java
      ├── CheckoutTopDownIntegrationTest.java
      ├── CustomerSearchIntegrationTest.java
      ├── EmployeeCRUDIntegrationTest.java
      ├── ProductCategoryIntegrationTest.java
      ├── ProductDALStub.java
      ├── ProductSupplierIntegrationTest.java
      ├── SaleDALStub.java
      ├── UserAuthIntegrationTest.java
      ├── UserIntegrationTest.java
      └── ValidationBottomUpIntegrationTest.java
  └── model.validators
      └── PasswordValidationTest.java
```



What was verified

- ✓ Correct interaction between **application logic, DAL, and MySQL database**
- ✓ Enforcement of **foreign key constraints** and referential integrity
- ✓ Proper error handling for invalid input and constraint violations
- ✓ Safe cleanup and restoration of database state after each test

System Testing



Performed by: Abdulaziz Bezan , Evisa Nela

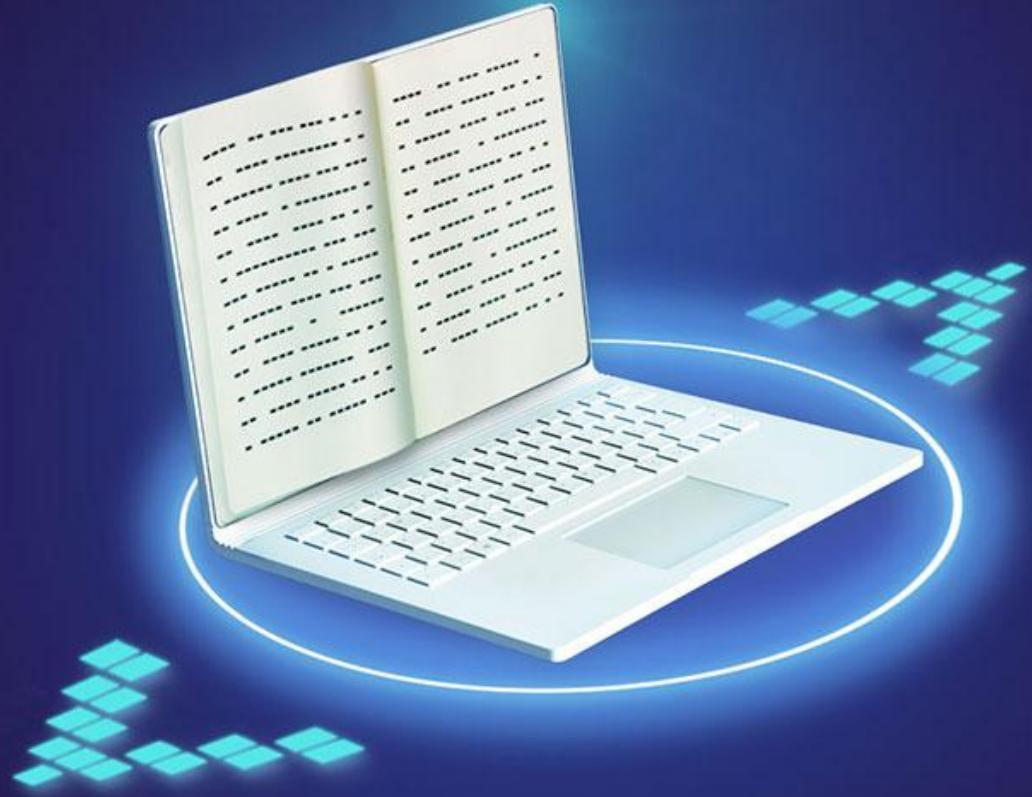


Evisa Nela

- Performed **end-to-end system testing** of the POS system
- Verified **user login authentication** across UI, controller, DAL, and database
- Tested **supplier deletion constraints**, ensuring foreign key enforcement
- Confirmed correct system behavior and data integrity
- **All system tests passed**

Abdulaziz Bezan

- Performed **end-to-end system testing** of all POS modules
- Found **critical failures** in UI-database integration and error handling
- Identified **missing input validation** causing crashes with invalid data
- **Test coverage analysis** revealed untested error paths (60% path coverage)
- Provided **specific fixes** for validation and exception handling
- Led to a system **FAILURE**, lack of error handling makes it less efficient.



THANK YOU ☺