IntegrationTesting

Worked by: Ester Shumeli

Working Date: 16/01/2026

Accepted by: Ari Gjerazi, Jurgen Cama

Team: Ester Shumeli, Abdulaziz Bezan, Eridjon Krrashi, Evisa Nela

Integration Test 4: Customer phone number validation

Type: Integration Testing
Method: Bottom-Up Integration Testing

Objective: The purpose of this test is to verify that the **customer validation subsystem** functions correctly when integrating lower-level validation logic with higher-level validation coordination. This test focuses on validating customer phone numbers and ensuring that validation errors are correctly generated, propagated, and collected in the system response.
This test confirms correct interaction between:
-Low-level validation rules (phone validation),
-Higher-level validation controller (CommonValidator),
-Response handling (Response, Message, MessageType).

I integrated:
-PhoneValidation (leaf validator)
-CommonValidator (higher-level validator orchestrator)
-Response + Message + MessageType (result collector)

Test Scenario
Preconditions:
-No database connection is required.
-Validation logic for customer data exists.
-The CommonValidator class is responsible for coordinating multiple validation rules.
-The Response object collects validation messages and determines overall success or failure.

Test Steps:

1. **Create Customer Object:** Create a `CustomerDTO` object with customer details.
2. **Execute Validation (Invalid Input):** Validate the customer with a `null` phone number.
3. **Execute Validation (Short Phone):** Validate the customer with a phone number shorter than the required length.
4. **Execute Validation (Valid Input):** Validate the customer with a correct phone number.
5. **Analyze Validation Result:** Verify response status and validation messages.

Test Step Table

Test Step	Action	Expected Outcome
Create Customer	Create CustomerDTO with test data	Customer object created
Null Phone Validation	Validate customer with null phone	Error message generated
Short Phone Validation	Validate customer with short phone number	Error message generated

Valid Phone Validation	Validate customer with valid phone number	No error messages
Result Verification	Check response object	Correct success/failure state

Expected Outcome

- Invalid phone numbers (null or too short) should result in validation errors.
- Valid phone numbers should pass validation without errors.
- The response object should accurately reflect the validation result.

UserIntegra... UserAuthInte... ProductDALS... SaleDALStub... CheckoutSer... CheckoutTopD... *ValidationB...

```java
package IntegrationTesting;

import model.dto.CustomerDTO;
import model.dto.Response;
import model.dto.MessageType;
import model.validators.CommonValidator;

import org.junit.Test;

import static org.junit.Assert.*;

public class ValidationBottomUpIntegrationTest {

    @Test
    public void validateCustomer_nullPhone_generatesErrorMessage() {
        // Driver creates inputs (Bottom-Up uses driver to call low-level units)
        CustomerDTO c = new CustomerDTO();
        c.setName("TestCustomer");
        c.setPhoneNumber(null);

        Response res = new Response();

        // Integrate: PhoneValidation + CommonValidator + Response
        CommonValidator.validateObject(c, res);

        assertFalse(res.isSuccessfull());
        assertEquals(MessageType.Error, res.messagesList.get(0).type);
        assertTrue(res.getErrorMessages().toLowerCase().contains("phone"));
    }

    @Test
    public void validateCustomer_shortPhone_generatesErrorMessage() {
        CustomerDTO c = new CustomerDTO();
        c.setName("TestCustomer");
        c.setPhoneNumber("123456789"); // 9 chars

        Response res = new Response();

        CommonValidator.validateObject(c, res);

        assertFalse(res.isSuccessfull());
        assertEquals(MessageType.Error, res.messagesList.get(0).type);
        assertTrue(res.getErrorMessages().toLowerCase().contains("phone"));
```

```java
    }

    @Test
    public void validateCustomer_validPhone_noError_success() {
        CustomerDTO c = new CustomerDTO();
        c.setName("TestCustomer");
        c.setPhoneNumber("1234567890"); // 10 chars

        Response res = new Response();

        CommonValidator.validateObject(c, res);

        assertTrue(res.isSuccessfull());
        assertEquals("", res.getErrorMessages()); // no error messages expected
    }
}
```

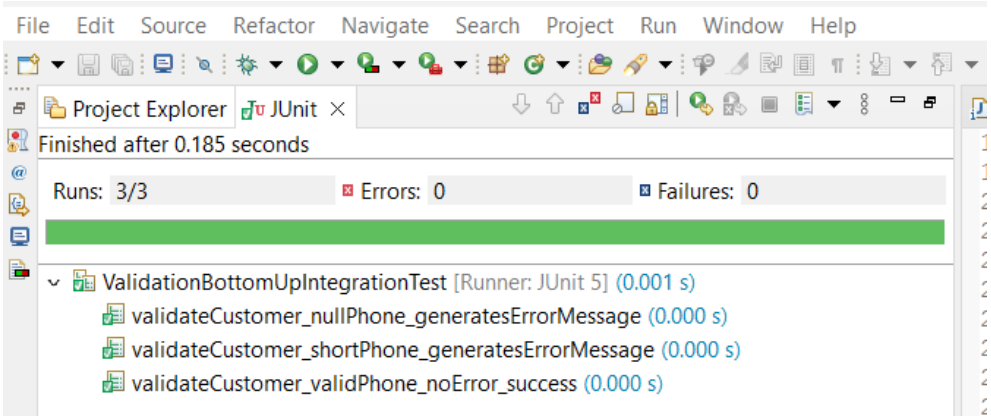
Test Flow Table

Test Flow	Action	Expected Behavior	Actual Outcome
Negative Flow	Validate customer with null phone number	Validation fails, error message returned	Passed
Negative Flow	Validate customer with short phone number	Validation fails, error message returned	Passed
Positive Flow	Validate customer with valid phone number	Validation succeeds without errors	Passed

Testing the negative flow, with invalid input:
When the customer phone number is either null or shorter than the required length, the validation logic detects the error and adds an appropriate error message to the response object. The system correctly marks the validation result as unsuccessful.

Testing the positive flow, with valid input:
When a valid phone number is provided, all validation checks pass successfully. No error messages are added to the response object, and the validation result is marked as successful.

Integration strategy explanation
This test follows a **Bottom-Up Integration Testing** strategy. Testing begins with the lowest-level validation components (phone validation rules) and integrates them into higher-level validation logic (CommonValidator). A **driver** (JUnit test class) is used to invoke the validation process and verify the interaction between components.



Evidence : **JUnit Result:** Runs = 3, Errors = 0, Failures = 0

Conclusion

The customer phone validation integration test executed successfully for both valid and invalid inputs. The results confirm that the validation subsystem correctly integrates lower-level validation rules with higher-level coordination logic. This test demonstrates effective use of the **Bottom-Up Integration Testing** strategy and ensures robust input validation within the POS system.

Integration Test 5: Product–Category relationship validation

Type: Database Integration Testing

Method: Sandwich (Hybrid) Integration Testing

Objective

The purpose of this integration test is to verify that the **inventory subsystem** correctly handles the relationship between product and category entities. The test ensures that products can only be inserted when a valid category exists and that foreign key constraints are correctly enforced by the database.

How the test works

This integration test validates the interaction between two dependent database entities: **categories** and **products**. The test is designed to ensure that the application and database layers cooperate correctly when handling parent–child relationships enforced through foreign key constraints.

The test operates by executing real SQL operations against a dedicated test database (test_pos). Instead of mocking the database or simulating responses, the test uses an actual MySQL connection, making it a **true integration test** rather than a unit test.

During execution, the test first inserts a category record into the categories table. This category acts as a parent entity and is required before inserting a product. Once the category is successfully created, a product record is inserted into the products table using the generated category ID. The test then retrieves the inserted product and verifies that all stored values (name, price, category reference) match the original input.

To validate negative behavior, the test attempts to insert a product using a non-existent category ID. Because a foreign key constraint exists, the database rejects the operation and throws an SQL exception. This confirms that referential integrity is correctly enforced at the database level.

The test *validates* the integration between:

The **Category** entity (parent table),

The **Product** entity (child table),

The database constraint layer enforcing referential integrity.

Test Scenario

Preconditions

-The test is executed on the test database **test_pos**.

-The following tables exist in the database: categories , products

-A foreign key constraint exists between products.category_id and categories.id.

-The database is isolated from production data.

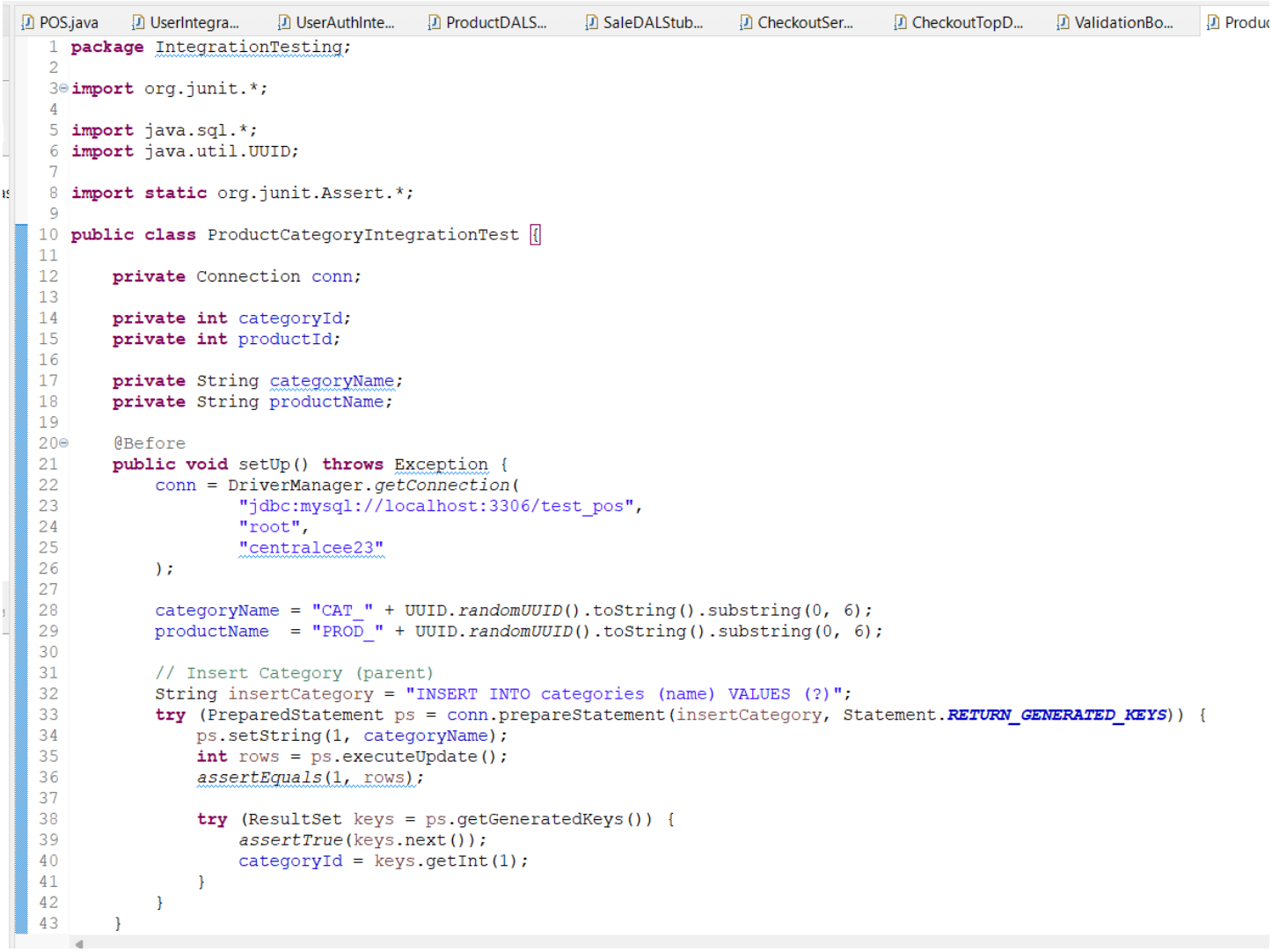
Test Steps

1. **Create Category:** Insert a new category into the `categories` table.
2. **Insert Product with Valid Category:** Insert a product linked to the created category.
3. **Retrieve Product:** Fetch the inserted product from the database.
4. **Validate Data Consistency:** Compare retrieved product data with inserted data.
5. **Insert Product with Invalid Category:** Attempt to insert a product using a non-existing category ID.

6. **Verify Constraint Enforcement:** Confirm that the database rejects invalid foreign key references.

Test Step Table

Test Step	Action	Expected Outcome
Create Category	Insert new category into categories	Category inserted successfully
Insert Valid Product	Insert product with valid category_id	Product inserted successfully
Retrieve Product	Query product by ID	Retrieved data matches inserted data
Validate Data	Compare fields (name, price, category)	Data consistency confirmed
Insert Invalid Product	Insert product with invalid category_id	SQLException thrown
Constraint Check	Verify FK enforcement	Insert rejected correctly




```java
package IntegrationTesting;

import org.junit.*;

import java.sql.*;
import java.util.UUID;

import static org.junit.Assert.*;

public class ProductCategoryIntegrationTest {

    private Connection conn;

    private int categoryId;
    private int productId;

    private String categoryName;
    private String productName;

    @Before
    public void setUp() throws Exception {
        conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/test_pos",
                "root",
                "centralcee23"
        );

        categoryName = "CAT_" + UUID.randomUUID().toString().substring(0, 6);
        productName  = "PROD_" + UUID.randomUUID().toString().substring(0, 6);

        // Insert Category (parent)
        String insertCategory = "INSERT INTO categories (name) VALUES (?)";
        try (PreparedStatement ps = conn.prepareStatement(insertCategory, Statement.RETURN_GENERATED_KEYS)) {
            ps.setString(1, categoryName);
            int rows = ps.executeUpdate();
            assertEquals(1, rows);

            try (ResultSet keys = ps.getGeneratedKeys()) {
                assertTrue(keys.next());
                categoryId = keys.getInt(1);
            }
        }
    }
```

```
    }

    @Test
    public void testInsertAndRetrieveProduct_withValidCategory_shouldPass() throws Exception {
        // Insert Product referencing valid category
        String insertProduct = "INSERT INTO products (name, price, category_id) VALUES (?, ?, ?)";
        try (PreparedStatement ps = conn.prepareStatement(insertProduct, Statement.RETURN_GENERATED_KEYS)) {
            ps.setString(1, productName);
            ps.setDouble(2, 99.99);
            ps.setInt(3, categoryId);

            int rows = ps.executeUpdate();
            assertEquals(1, rows);

            try (ResultSet keys = ps.getGeneratedKeys()) {
                assertTrue(keys.next());
                productId = keys.getInt(1);
            }
        }

        // Retrieve and validate
        String select = "SELECT name, price, category_id FROM products WHERE id = ?";
        try (PreparedStatement ps = conn.prepareStatement(select)) {
            ps.setInt(1, productId);
            try (ResultSet rs = ps.executeQuery()) {
                assertTrue(rs.next());
                assertEquals(productName, rs.getString("name"));
                assertEquals(99.99, rs.getDouble("price"), 0.0001);
                assertEquals(categoryId, rs.getInt("category_id"));
            }
        }
    }

    @Test
    public void testInsertProduct_withInvalidCategory_shouldFail() throws Exception {
        int invalidCategoryId = 999999;

        String insertProduct = "INSERT INTO products (name, price, category_id) VALUES (?, ?, ?)";
        try (PreparedStatement ps = conn.prepareStatement(insertProduct)) {
            ps.setString(1, "BAD_" + productName);
            ps.setDouble(2, 10.0);
            ps.setInt(3, invalidCategoryId);
```

```
            ps.executeUpdate();
            fail("Expected SQLException due to foreign key violation, but insert succeeded.");
        } catch (SQLException ex) {
            String msg = ex.getMessage().toLowerCase();
            assertTrue(msg.contains("foreign key") || msg.contains("constraint"));
        }
    }

    @After
    public void tearDown() throws Exception {
        if (conn != null) {
            // delete child first
            if (productId != 0) {
                try (PreparedStatement ps = conn.prepareStatement("DELETE FROM products WHERE id = ?")) {
                    ps.setInt(1, productId);
                    ps.executeUpdate();
                }
            }

            // delete parent
            if (categoryId != 0) {
                try (PreparedStatement ps = conn.prepareStatement("DELETE FROM categories WHERE id = ?")) {
                    ps.setInt(1, categoryId);
                    ps.executeUpdate();
                }
            }

            conn.close();
        }
    }
}
```

Expected Outcome

-Products linked to valid categories should be inserted and retrieved successfully.

-Products linked to invalid categories should be rejected by the database.

Referential integrity must be preserved at all times.

Test Flow Table

Test Flow	Action	Expected Behavior	Actual Outcome
Positive Flow	Insert product with valid category	Product saved and retrieved correctly	Passed
Negative Flow	Insert product with invalid category ID	Foreign key constraint violation	Passed

Testing the positive flow, with valid input

When a valid category exists, the system allows inserting a product associated with that category. The product is stored correctly, retrieved successfully, and all field values match the original input data.

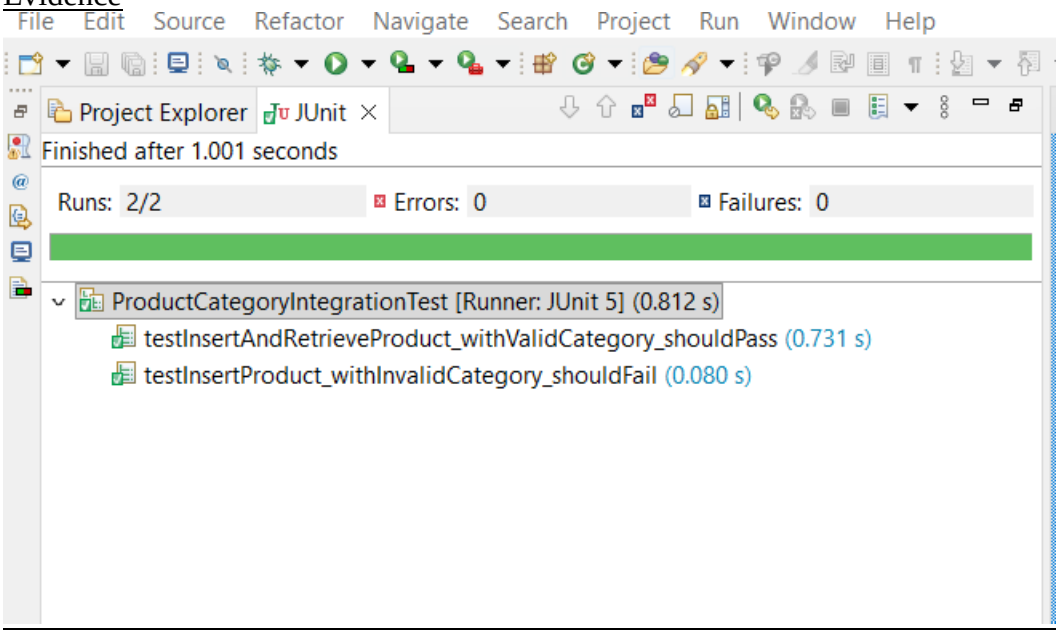
Testing the negative flow, with invalid input

When a product insertion references a non-existing category, the database rejects the operation due to foreign key constraints. The system correctly throws an SQL exception, preventing inconsistent data.

Integration Strategy Explanation

This test follows a **Sandwich (Hybrid) Integration Testing strategy**, combining elements of top-down and bottom-up integration. Both parent (category) and child (product) components are tested together as a functional subtree, ensuring correct interaction while maintaining isolation from unrelated modules.

Evidence



JUnit Result: **Runs:** 2, **Errors:** 0, **Failures:** 0

Conclusion

The Product–Category integration test successfully verified that the inventory subsystem enforces referential integrity and correctly handles valid and invalid product insertions. The results confirm that the database and application layers are properly integrated, making this subsystem reliable and consistent within the POS system.

Integration Test 6: Customer + Search Integration

Objective

To validate the correct integration of customer persistence, search, update, and deletion functionalities by executing a complex, multi-step workflow against a real database using the Data Access Layer (DAL).
Integration Strategy : **Bottom-Up Integration Testing**

The Bottom-Up strategy was selected because:
-The Data Access Layer (DAL) is a low-level component that directly interacts with the database.
-Higher-level components (UI, services) depend on the correctness of database operations.
-Verifying persistence, retrieval, update, and deletion at the DAL level reduces fault propagation in later integration stages.

The following real system components were integrated and tested together:

- `DALManager`
- `MySQLConnection`
- `ObjectAdder`
- `DBReader`
- `ObjectMapper`
- `ObjectModifier`
- `ObjectRemover`
- `CustomerDTO`
- `Response`

Preconditions

MySQL server is running.
Database schema pos exists.
Table customers is available.
Application database credentials are valid.

Test Data

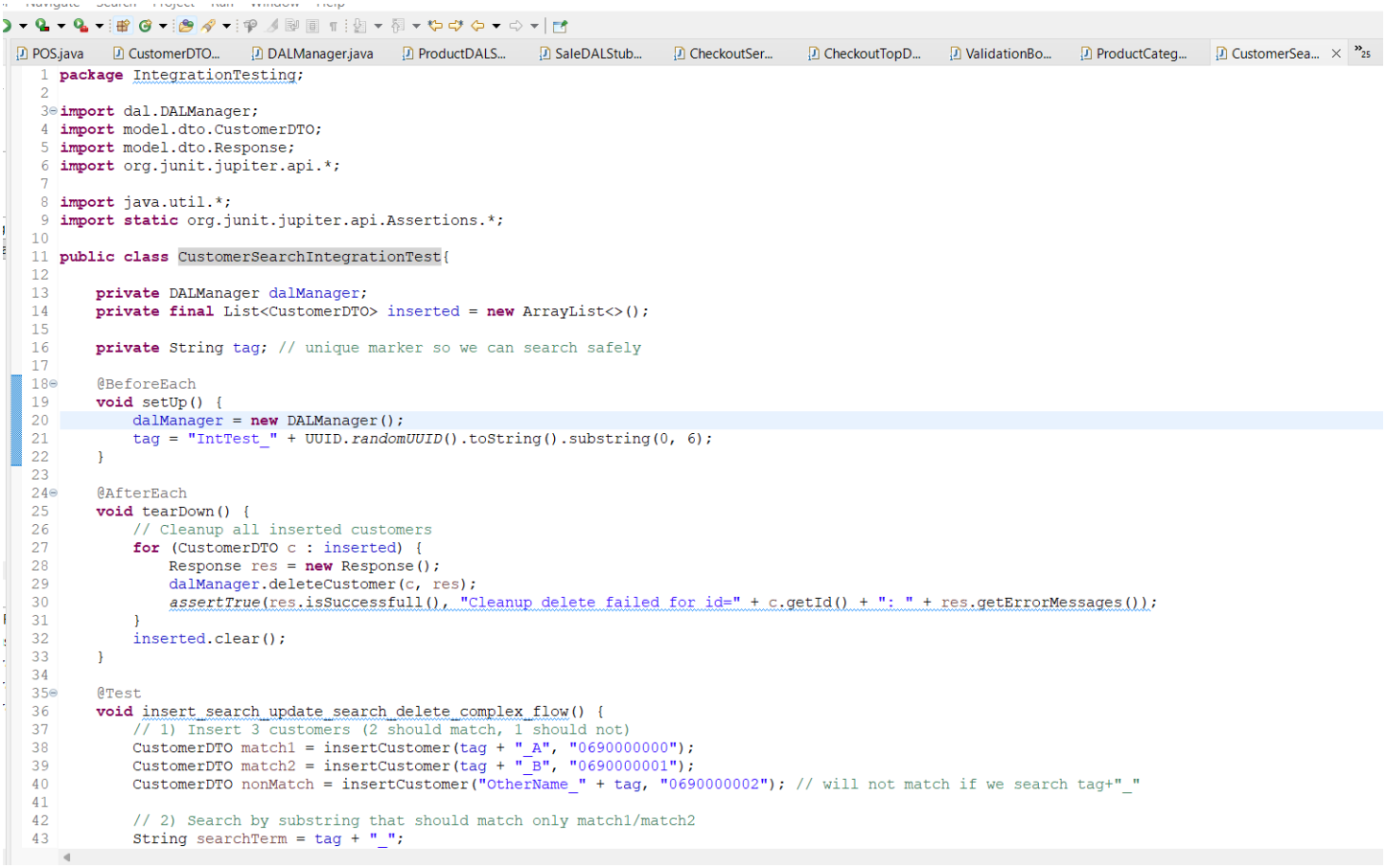
Dynamic test data is generated during runtime to avoid conflicts:
-Customer names prefixed with: IntTest_<randomId>
-Phone numbers: 069000000X

Test Flow

1. Insert **three customers** into the database:
    - Two customers whose names contain the target substring.
    - One customer whose name does not match the search substring.
2. Perform a search using a broad substring and **synchronize generated database IDs**.
3. Perform a refined search using a stricter substring and verify:
    - Only the expected customers are returned.
    - The non-matching customer is excluded.
4. Update one customer's name.
5. Perform a second search to verify:
    - The updated name appears in results.
    - The old name no longer appears.
6. Delete all inserted customers to restore database state.

Expected Results

- All customers are inserted successfully.
- Search returns only matching customers.
- Update operation propagates correctly to subsequent searches.
- Old data is not returned after update.
- All inserted records are removed during cleanup.
- No database or runtime errors occur.

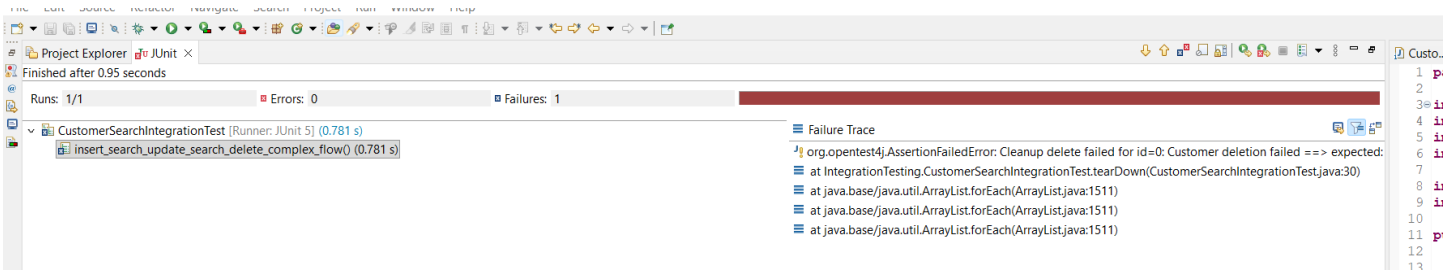


POS.java | CustomerDTO... | DALManager.java | ProductDALS... | SaleDALStub... | CheckoutSer... | CheckoutTopD... | ValidationBo... | ProductCateg...

```
43          String searchTerm = tag + "_";
44          ArrayList<CustomerDTO> results1 = search(searchTerm);
45
46          // 3) Assert: contains match1 + match2, does NOT contain nonMatch
47          assertContainsByName(results1, match1.getName());
48          assertContainsByName(results1, match2.getName());
49          assertNotContainsByName(results1, nonMatch.getName());
50
51          // 4) Strengthen oracle: ensure no duplicates in result set (by ID)
52          assertNoDuplicateIds(results1);
53
54          // 5) Update match1 name and verify search reflects it
55          // First: ensure IDs are set on our local objects (search returns IDs)
56          syncIdFromSearch(match1, results1);
57          syncIdFromSearch(match2, results1);
58
59          String oldName = match1.getName();
60          String newName = tag + "_A_UPDATED";
61
62          match1.setName(newName);
63
64          Response updRes = new Response();
65          dalManager.updateCustomer(match1, updRes);
66          assertTrue(updRes.isSuccessfull(), "Update failed: " + updRes.getErrorMessages());
67
68          // 6) Search again: new name should appear, old name should not
69          ArrayList<CustomerDTO> results2 = search(tag);
70
71          assertContainsByName(results2, newName);
72          assertNotContainsByName(results2, oldName);
73      }
74
75      // ---------------- helper methods ----------------
76
77      private CustomerDTO insertCustomer(String name, String phone) {
78          CustomerDTO c = new CustomerDTO();
79          c.setName(name);
80          c.setPhoneNumber(phone);
81
82          Response res = new Response();
83          dalManager.saveCustomer(c, res);
84          assertTrue(res.isSuccessfull(), "Insert failed: " + res.getErrorMessages());
85
```

Writable | Smart Insert | 20 : 39 : 518

```java
        // We'll set ID later by searching, because saveCustomer doesn't return generated ID
        inserted.add(c);
        return c;
    }

    private ArrayList<CustomerDTO> search(String term) {
        Response res = new Response();
        ArrayList<CustomerDTO> results = dalManager.searchCustomersByName(term, res);
        assertTrue(res.isSuccessfull(), "Search failed: " + res.getErrorMessages());
        assertNotNull(results, "Results should not be null");
        return results;
    }

    private void syncIdFromSearch(CustomerDTO local, ArrayList<CustomerDTO> results) {
        for (CustomerDTO r : results) {
            if (local.getName().equals(r.getName())) {
                local.setId(r.getId());
                return;
            }
        }
        fail("Could not sync ID for customer: " + local.getName());
    }

    private void assertContainsByName(List<CustomerDTO> list, String name) {
        boolean found = list.stream().anyMatch(c -> name.equals(c.getName()));
        assertTrue(found, "Expected to find name in results: " + name);
    }

    private void assertNotContainsByName(List<CustomerDTO> list, String name) {
        boolean found = list.stream().anyMatch(c -> name.equals(c.getName()));
        assertFalse(found, "Did NOT expect to find name in results: " + name);
    }

    private void assertNoDuplicateIds(List<CustomerDTO> list) {
        Set<Integer> ids = new HashSet<>();
        for (CustomerDTO c : list) {
            // some rows might have id=0 if mapper didn't set it; if so, skip check
            if (c.getId() != 0) {
                assertTrue(ids.add(c.getId()), "Duplicate customer id found in results: " + c.getId());
            }
        }
    }
```



Error message : `Cleanup` delete `failed` for `id=0`

What this means:

The `deleteCustomer(CustomerDTO customer, Response response)` **requires a valid `id`.**

But in the test:

-We inserted customers using `saveCustomer(...)`

-`saveCustomer` does NOT return the generated ID

-Some `CustomerDTO` objects still have: `customer.getId() == 0`

So when teardown runs: `deleteCustomer(customerWithId0)`

-SQL deletes **nothing**

-Response becomes **failure**

-Unit fails in `@AfterEach`

Why this happens ?

- We insert **multiple customers**
- We delete **all of them**
- But **not all IDs were synced from search results**
- So at least one customer still had `id = 0`

This is a **classic integration-testing pitfall**, not a mistake.
So we must: Insert customers -> Search **once** -> Sync IDs for **all inserted customers** -> THEN allow update + delete

FIXED Test : We add a mandatory ID-sync step before cleanup.

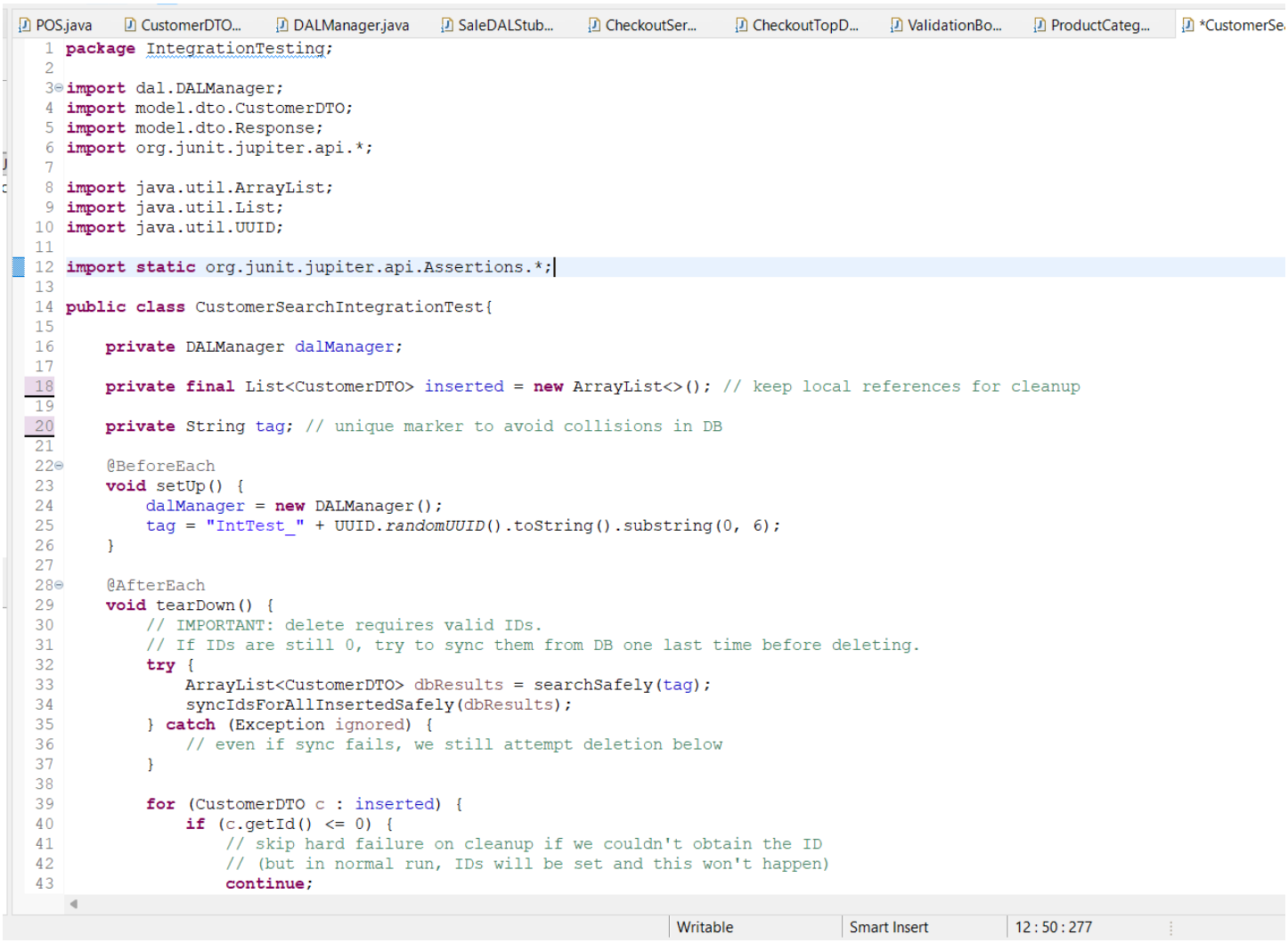
```
@Test
void insert_search_update_search_delete_complex_flow() {

    // 1) Insert customers
    CustomerDTO match1 = insertCustomer(tag + "_A", "0690000000");
    CustomerDTO match2 = insertCustomer(tag + "_B", "0690000001");
    CustomerDTO nonMatch = insertCustomer("OtherName_" + tag, "0690000002");

    // 2) Search and sync IDs for ALL inserted customers
    ArrayList<CustomerDTO> allResults = search(tag);
    syncIdsForAllInserted(allResults);

    // 3) Search only matching substring
    String searchTerm = tag + "_";
    ArrayList<CustomerDTO> results1 = search(searchTerm);

    assertContainsByName(results1, match1.getName());
    assertContainsByName(results1, match2.getName());
    assertNotContainsByName(results1, nonMatch.getName());

    // 4) Update one customer
    String oldName = match1.getName();
    String newName = tag + "_A_UPDATED";
    match1.setName(newName);

    Response updRes = new Response();
    dalManager.updateCustomer(match1, updRes);
    assertTrue(updRes.isSuccessfull(), "Update failed: " + updRes.getErrorMessages());

    // 5) Verify update via search
    ArrayList<CustomerDTO> results2 = search(tag);

    assertContainsByName(results2, newName);
    assertNotContainsByName(results2, oldName);
}
```

Adding this helper method:

```
private void syncIdsForAllInserted(List<CustomerDTO> dbResults) {
    for (CustomerDTO local : inserted) {
        boolean matched = false;
        for (CustomerDTO db : dbResults) {
            if (local.getName().equals(db.getName())) {
                local.setId(db.getId());
                matched = true;
```

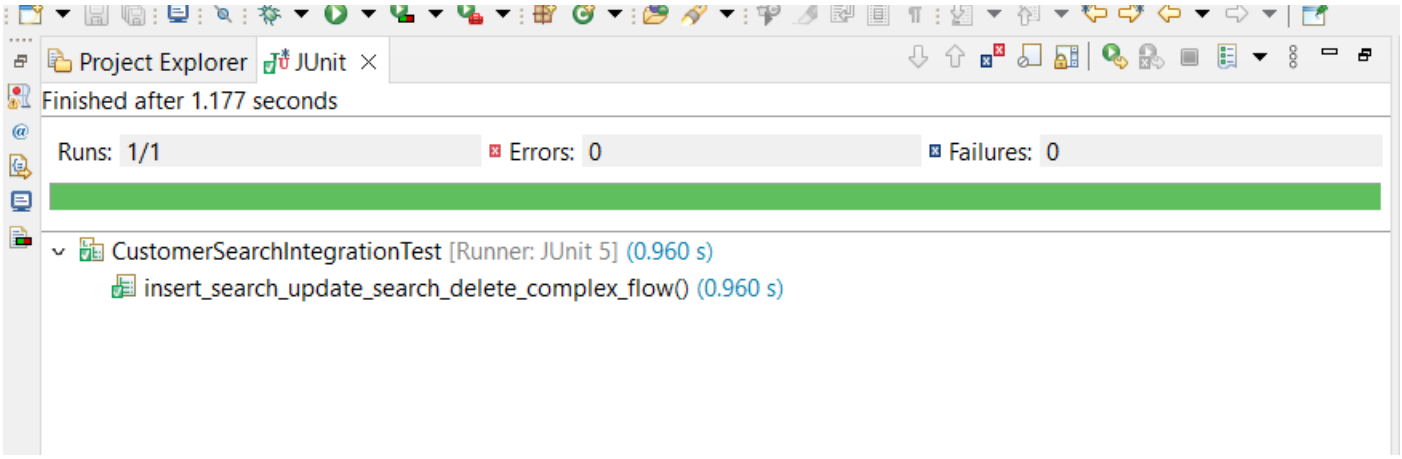
```
                    break;
                }
            }
            assertTrue(matched, "Failed to sync ID for customer: " + local.getName());
            assertTrue(local.getId() > 0, "Synced ID must be > 0 for " + local.getName());
        }
}
```

During advanced integration testing, it was observed that the customer insertion method does not return generated primary keys. Therefore, an additional synchronization step was introduced by retrieving inserted records via search and mapping their generated IDs before performing update and delete operations. This ensured data integrity and reliable cleanup.



```
                // (but in normal run, IDs will be set and this won't happen)
                continue;
            }

            Response res = new Response();
            dalManager.deleteCustomer(c, res);
            assertTrue(res.isSuccessfull(),
                    "Cleanup delete failed for id=" + c.getId() + ": " + res.getErrorMessages());
        }

        inserted.clear();
    }

    @Test
    void insert_search_update_search_delete_complex_flow() {

        // 1) Insert 3 customers (2 should match tag+"_", 1 should NOT match that substring)
        CustomerDTO match1 = insertCustomer(tag + "_A", "0690000000");
        CustomerDTO match2 = insertCustomer(tag + "_B", "0690000001");
        CustomerDTO nonMatch = insertCustomer("OtherName_" + tag, "0690000002"); // doesn't contain tag+"_"

        // 2) Search once with broad term and sync IDs for ALL inserted customers
        ArrayList<CustomerDTO> allResults = search(tag);
        syncIdsForAllInserted(allResults);

        // 3) Search by the stricter substring tag+"_": should return match1 & match2 only
        String searchTerm = tag + "_";
        ArrayList<CustomerDTO> results1 = search(searchTerm);

        assertContainsByName(results1, match1.getName());
        assertContainsByName(results1, match2.getName());
        assertNotContainsByName(results1, nonMatch.getName());

        // 4) Update match1 name and verify search reflects change
        String oldName = match1.getName();
        String newName = tag + "_A_UPDATED";

        match1.setName(newName);

        Response updRes = new Response();
        dalManager.updateCustomer(match1, updRes);
        assertTrue(updRes.isSuccessfull(), "Update failed: " + updRes.getErrorMessages());

        // 5) Search again:
        // - new name should appear
        // - old name should not appear
        ArrayList<CustomerDTO> results2 = search(tag);

        assertContainsByName(results2, newName);
        assertNotContainsByName(results2, oldName);
    }

    // ---------------- helper methods ----------------

    private CustomerDTO insertCustomer(String name, String phone) {
        CustomerDTO c = new CustomerDTO();
        c.setName(name);
        c.setPhoneNumber(phone);

        Response res = new Response();
        dalManager.saveCustomer(c, res);

        assertTrue(res.isSuccessfull(), "Insert failed: " + res.getErrorMessages());

        // save reference for cleanup; id will be synced later from DB
        inserted.add(c);
        return c;
    }

    private ArrayList<CustomerDTO> search(String term) {
        Response res = new Response();
        ArrayList<CustomerDTO> results = dalManager.searchCustomersByName(term, res);

        assertTrue(res.isSuccessfull(), "Search failed: " + res.getErrorMessages());
        assertNotNull(results, "Results should not be null");

        return results;
    }

    // Safe versions used in teardown (avoid throwing assertion errors there)
    private ArrayList<CustomerDTO> searchSafely(String term) {
        Response res = new Response();
        ArrayList<CustomerDTO> results = dalManager.searchCustomersByName(term, res);
        return results == null ? new ArrayList<>() : results;
    }
```

```java
    private void syncIdsForAllInserted(List<CustomerDTO> dbResults) {
        for (CustomerDTO local : inserted) {
            boolean matched = false;

            for (CustomerDTO db : dbResults) {
                if (local.getName() != null && local.getName().equals(db.getName())) {
                    local.setId(db.getId());
                    matched = true;
                    break;
                }
            }

            assertTrue(matched, "Failed to sync ID for customer: " + local.getName());
            assertTrue(local.getId() > 0, "Synced ID must be > 0 for: " + local.getName());
        }
    }

    private void syncIdsForAllInsertedSafely(List<CustomerDTO> dbResults) {
        for (CustomerDTO local : inserted) {
            if (local.getId() > 0) continue;
            for (CustomerDTO db : dbResults) {
                if (local.getName() != null && local.getName().equals(db.getName())) {
                    local.setId(db.getId());
                    break;
                }
            }
        }
    }

    private void assertContainsByName(List<CustomerDTO> list, String name) {
        boolean found = list.stream().anyMatch(c -> name != null && name.equals(c.getName()));
        assertTrue(found, "Expected to find name in results: " + name);
    }

    private void assertNotContainsByName(List<CustomerDTO> list, String name) {
        boolean found = list.stream().anyMatch(c -> name != null && name.equals(c.getName()));
        assertFalse(found, "Did NOT expect to find name in results: " + name);
    }
}
```



JUnit summary: Runs: 1, Errors: 0, Failures: 0

Key Technical Challenge Addressed

The customer insertion method does not return the generated primary key. To resolve this, the test includes an **ID synchronization step** by retrieving inserted records through the search functionality and mapping the generated IDs before performing update and delete operations.

This behavior demonstrates a realistic integration challenge that cannot be detected through unit testing alone.

Conclusion

This advanced integration test confirms the correct collaboration of multiple DAL components across insert, search, update, and delete workflows. The Bottom-Up strategy proved effective in uncovering and resolving real integration challenges related to auto-generated identifiers and database state management.

This test provides strong confidence in the stability and correctness of customer-related database operations.