**Verificação e Validação de Software 24/25**

**Assignment 2 – Report**

**Tiago de Almeida – fc58161**

**Introduction**

This report documents the testing and analysis activities that I was tasked with for this assignment. The main objective was to apply software integration testing tools to a web application structured in a layered architecture, composed of a persistence layer, business layer and a web presentation layer. The *MVC* pattern was used in the construction of the system.

Using tools like ***HtmlUnit***, ***DbSetup*** and ***Mockito***, the goal was to validate the functional behavior of the *WebApp*, ensure database consistency and evaluate the feasibility of isolating business logic using mocks.

The contents of this report include the development of automated test scenarios using ***JUnit***, database setup procedures, module-level testing with mock dependencies and the identification and reporting of *bugs* / *faults*. This contributes to the overall quality assurance process of the *SUT*.

**Test Environment Setup**

To effectively validate the behavior of the *WebApp*, a robust test environment was established using integration and unit testing tools appropriate for enterprise *Java* applications.

The application is deployed and tested on the **Wildfly** **application server**, a lightweight *Java EE* runtime that supports *servlets*, *JSPs* and *enterprise beans*.

The following tools were integrated:

* **HtmlUnit:** A headless browser testing framework used to simulate user interactions with the web interface. This allowed for validation of the *UI* behavior and the correct execution of use cases such as inserting customers, creating sales and navigating between views.
* **DbSetup:** A database preparation and reset library used to initialize the test database with known states. This ensured that tests could be executed with predictable data and validated against expected behaviors, such as constraints on duplicate *VAT* numbers or cascading deletes.
* **Mockito:** A mocking framework used to isolate and unit test the business layer. Where feasible, services that depended on other modules were decoupled through mocking to allow independent verification of their logic and behavior.

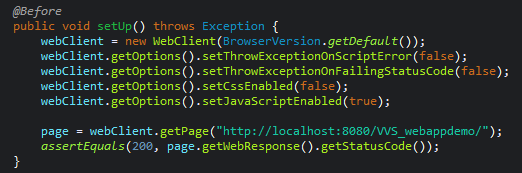
All tests were developed as *JUnit* test cases and integrated into the provided *Maven* project’s ***src/test/java*** folder. The ***pom.xml*** contains the necessary dependencies for the tests and app to work as intended.

**HtmlUnit Tests**

**Test Case 1 - Insert two new addresses for an existing customer, then the table of addresses of that client includes those addresses and its total row size increases by two**

*(The code for this test case is present in the* ***src/test/java/vvs\_htmlunit/TestInsertNewAddresses.java*** *file)*

Before even starting the test, I use the *@Before* notation, to setup the webclient to be used in the test, starting by asserting that, when accessing the page, a 200 OK code is thrown, meaning that the app is accessible.



To select an existing customer, I choose to access the page that lists all of the customers, then get the table that is present in the page. After verifying that it is the correct table, by checking the first row (contains the designations of each column), I assert that the table exists, and then select the VAT of the customer in the first row, by selecting the third cell.

After this, I execute the same process of getting the table, but on the page that lists the information about a specific customer by using its VAT. Then, if the table exists (meaning the user already has inserted addresses), we save the number of rows present in the table (minus the first row, that contains the designations of each column).



Then, I insert two new addresses into the customer’s data, using the **insertAddress** helper function. In the function, I start by accessing the **addAddressToCustomer.html** page and ensure that the page’s title is correct, then fill the fields with the desired address’ specifications and finally submit the form, that inserts the address into the database.

The landing page after submitting is the specified customer’s info page, that should now contain the new address on its address table. I execute the same process of getting a table used before, asserting that it is not *null*, and finally ensure that the last row’s contents correspond to the ones of the address we just added.

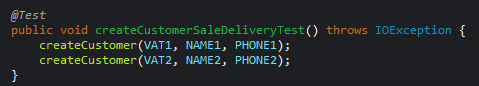
Lastly, I execute again the process of getting the table on the page that lists the information about a specific customer by using its VAT. Then, I ensure that the table exists and then ensure that the number of rows of this last table is equal to the number of recorded rows of the previous table plus two.



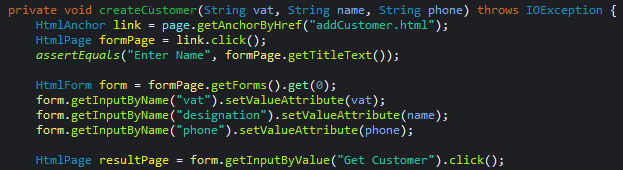
**Test Case 2 – Insert two new customers and check if all the information is properly listed in the *List All Customers* use case**

*(The code for this test case is present in the* ***src/test/java/vvs\_htmlunit/TestInsertNewCustomers.java*** *file)*

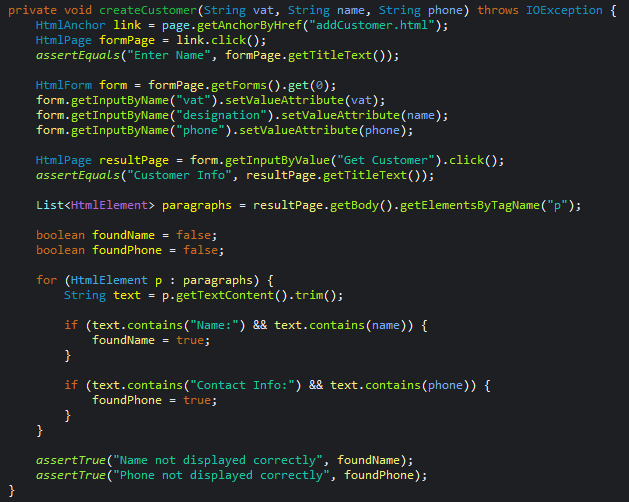
Firstly, I execute the setup process, the same as the prior test case. For this test, I just add two customers, using the createCustomer helper function.



In this function, I start by checking the correctness of the addCustomer.html page, then filling the required fields with the chosen customer’s data, and finally clicking the submit button.



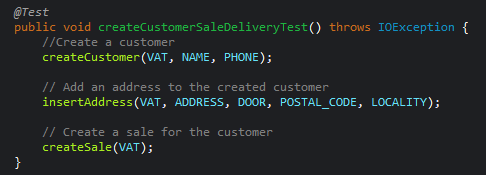
The landing page is the page that lists the information about the customer we just added. I start by checking the correctness of the title, then get the lines that state the name and phone number of the customer that was just added. For the name field, there is a Boolean variable that is changed to true if the found paragraph contains “Name:” and the name that we chose for the customer. The same is verified for the phone number field, using another Boolean variable that checks that a paragraph that contains “Contact Info:” and the phone number that we just added. Finally, we ensure that both variables were set to *True* at some point. After the test finishes, the added customers are removed from the database.



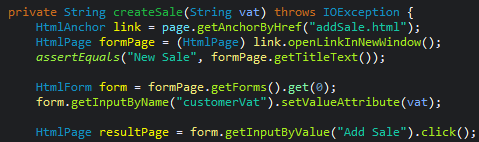
**Test Case 3 – A new sale will be listed as an open sale for the respective customer**

*(The code for this test case is present in the* ***src/test/java/vvs\_htmlunit/TestNewSaleState.java*** *file)*

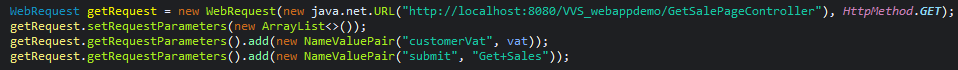
Firstly, I execute the setup process, the same as the prior test cases. Before creating the sale, the same **createCustomer** and **insertAddress** helper functions that we used in the prior test cases are used to create a customer and an address for them. After that, we create a sale using the **createSale** helper function.



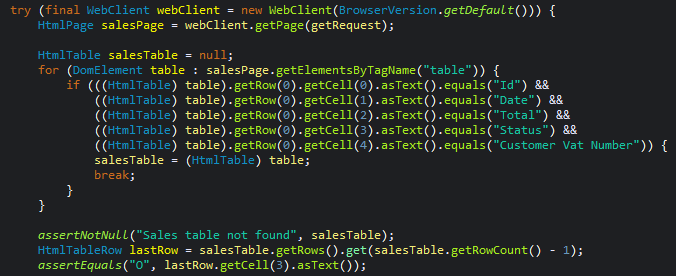
For this helper function, I start by accessing the **addSale.html** page, and check if the title’s correctness. After that, I fill the required VAT field with the client’s VAT that was just added and submit.



Then, I access the page that lists the sales of a certain customer by its VAT, by changing the URL parameters to the ones that correspond to the customer (VAT, method).



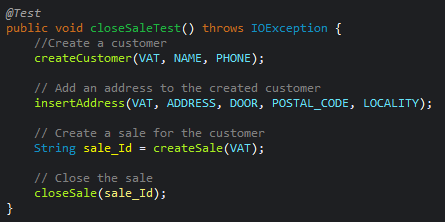
After this, I execute the same process of getting the table that contains the sales created for the customer, asserting that it is not null. Finally, I check that the last row (the sale we just created) has in its fourth cell (Sale’s Status) the letter “O”, meaning that it is an open sale. After the test finishes, the added customer is removed from the database.



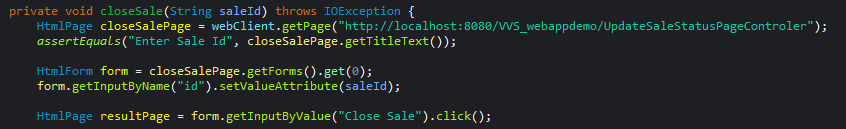
**Test Case 4 – After closing a sale, it will be listed as closed**

*(The code for this test case is present in the* ***src/test/java/vvs\_htmlunit/TestCloseSale.java*** *file)*

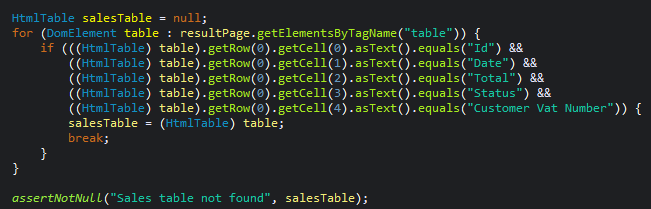
I started by setting up the **WebClient** like in the previous tests. Before closing the sale, the **createCustomer**, **insertAddress** and **createSale** helper functions are used to create a customer and a sale. After that, the **closeSale** helper function is used to close the sale.



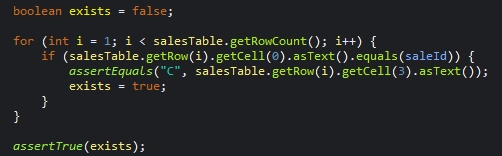
This function starts by accessing the page that updates the status of sales, then ensuring the correctness of the page’s title. After that, the form is filled with the sale ID that we got from the **createSale** helper function. Then, the form is submitted.



The result page is the updated page with the new state of the sales after closing the selected sale, so we utilize the same process of getting a table for the sales table, ensuring that it is not null.



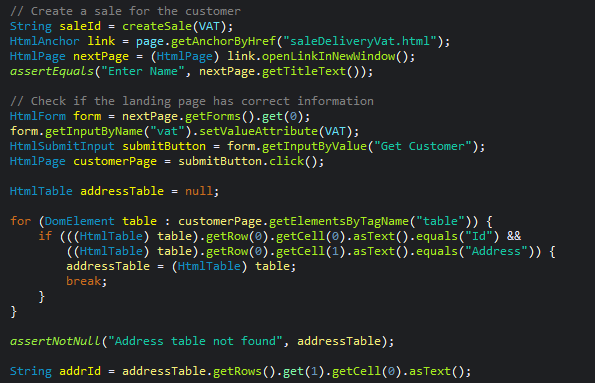
Finally, I use a for loop to iterate through all the rows in the sales table. If a row is found that has the ID of the added sale, we ensure that the fourth cell’s text (Sale status) is equal to “C”, meaning that is closed. I also use a *Boolean* variable, to ensure that the created sale was present on the table. After the test finishes, the added customer is removed from the database.



**Test Case 5 – Create a new customer, create a new sale for her, insert a delivery for that sale and then show the sale delivery. Check that all intermediate pages have the expected information**

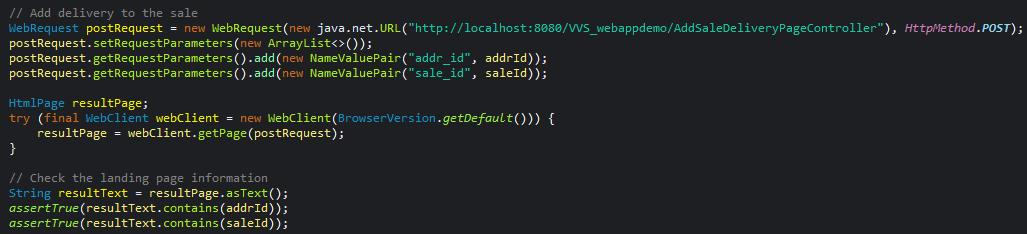
*(The code for this test case is present in the* ***src/test/java/vvs\_htmlunit/TestNewSaleDelivery.java*** *file)*

Firstly, I execute the setup process, the same as the prior test cases. Before creating the delivery, the same **createCustomer** and **insertAddress** helper functions that we used in the prior test cases are used to create a customer and an address for them. After that, we create a sale using the **createSale** helper function, saving the resulting ID of the sale. After that, I access the address table and select the ID of the first address.

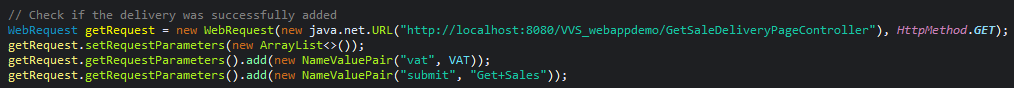


Then, I created a post request to the page that adds the delivery, manipulating the URL to include the sale ID and address ID that were saved previously.

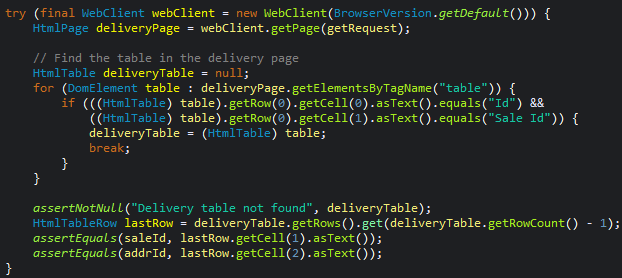
In the page that results from this submission, I start by checking if the address and sale IDs are present.



I access the page that lists the sale deliveries for a specific VAT, manipulating the URL to include the VAT and the method.



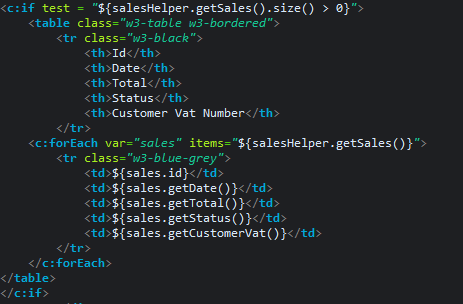
Finally, after asserting that the table that contains the deliveries exists, I access it and ensure that the last row (corresponding to the one we just added) contains the sale ID and address ID that we selected. After the test finishes, the added customer is removed from the database.



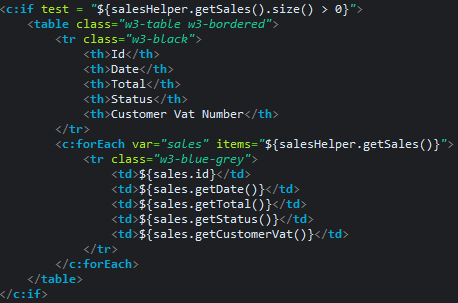
**Found faults/bugs and fixes:**

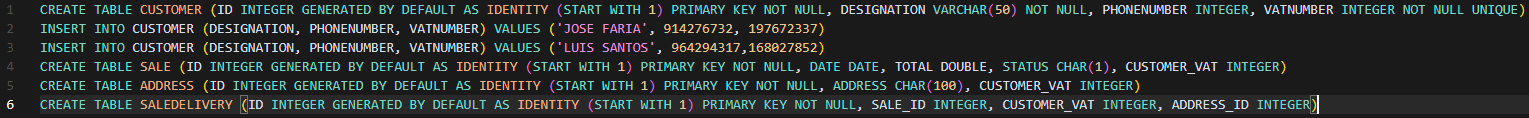
When developing the tests, some bugs were found that needed correction to make the intended tests work. They were as follows:

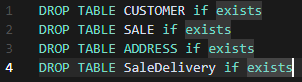
1. In the fourth test case (**Close Sale**), when executing the process of getting the table that was present on the page that updates the sales’ status, the code was selecting the lines that contained the *CSS* styles of the table, not the rows that contained the information about the sale. This happened because the **CloseSale.jsp** page was wrongly formatted. After fixing the formatting, the code started working as intended.
   1. **Original Code:**

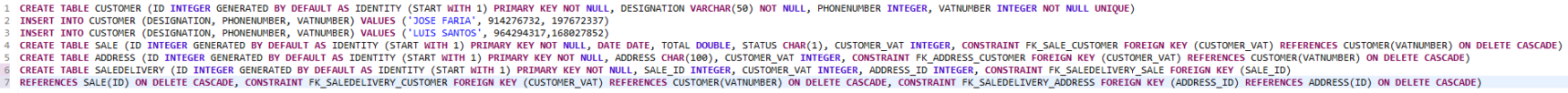


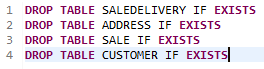
* 1. **Changed Code:**



1. After the tests were finished, deleting the added users, addresses or sales that were created was needed, so, the page that removes a customer by its VAT was used. Even though the customer was being deleted properly from the database, the rest of the created entries were not being deleted. This happened because the script that created the database (**src/main/webapp/CreateDatabase.java**) used the *.sql* create and drop scripts present in the **src/resources** folder, and those two scripts did not contain statements of the type “*ON DELETE CASCADE*”, which meant that even if the other information was associated to the customer’s information, it would not be deleted and would stay orphaned. After fixing the *SQL* code and running it again, everything worked as intended. Also, because the information would be linked to the customer entry, we would also need to change the order of the *DROP*s, so that the customer table would be the last table dropped, avoiding conflicts.
   1. **Original Code:**



* 1. **Changed Code (a few extra line breaks were inserted for the sake of presentation, the actual code in the file has each instruction in a single line in order for it to work properly):**



**DbSetup Tests**

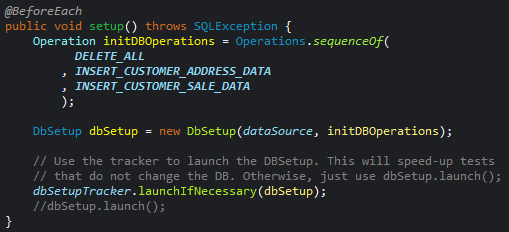
**Setup Class and Test Setup:**

For the setup, the **DBSetupUtils** class present in the materials of week 11 was used. Adjustments were made in the ***INSERT\_CUSTOMER\_SALE\_DATA*** operation, because it duplicated the i**nsertCustomers** *Insert*, causing errors and conflicts. The rest of the class was left unchanged.



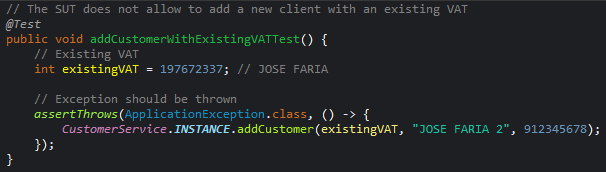
(Removed **insertCustomers**)

Using the **@BeforeEach** tag, before the start of each test all data is deleted and inserted again, using the operations in the **DBSetupUtils** class.



**Test Case 1 – The SUT does not allow to add a new client with an existing VAT**

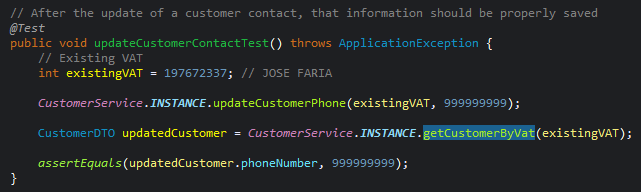
*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***addCustomerWithExistingVATTest()*** *function)*

For this test, I start by choosing an existing VAT that was already present in the database, then use an instance of **CustomerService** to invoke **addCustomer** with the chosen *VAT*, ensuring that an exception is thrown.

**Test Case 2 – After the update of a customer contact, that information should be properly saved**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***updateCustomerContactTest()*** *function)*

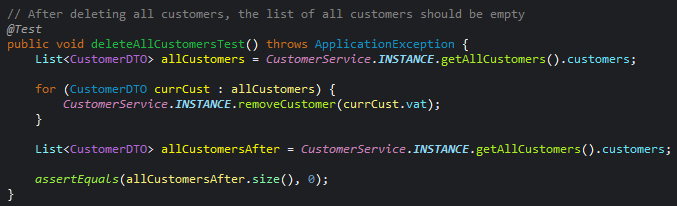
For this test, I start by choosing an existing *VAT* that was already present in the database, then use an instance of **CustomerService** to invoke **updateCustomerPhone** with the chosen *VAT*. After that, I use an instance of **CustomerService** to invoke **getCustomerByVat** with the chosen *VAT* and ensure that the retrieved customer’s phone number matches the one chosen to be the new contact.



**Test Case 3 – After deleting all customers, the list of all customers should be empty**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***deleteAllCustomersTest()*** *function)*

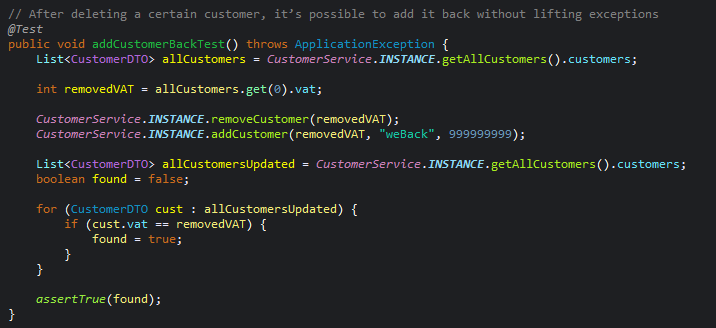
For this test, I start by using an instance of **CustomerService** to invoke **getAllCustomers**. After that, I iterate the returned list, using an instance of **CustomerService** to invoke **removeCustomer** for each *VAT*. Finally, I retrieve all customers again and ensure that the returned list’s size is equal to 0.



**Test Case 4 – After deleting a certain customer, it’s possible to add it back without lifting exceptions**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***addCustomerBackTest()*** *function)*

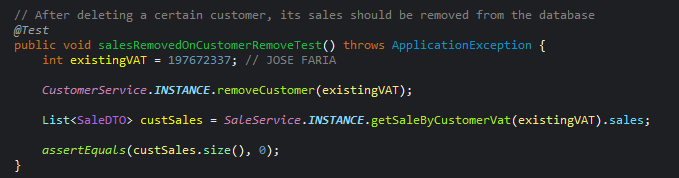
For this test, I start by using an instance of **CustomerService** to invoke **getAllCustomers**. After that, I choose the first customer in the list and use an instance of **CustomerService** to invoke **removeCustomer** with their *VAT*, and immediately after invoking **addCustomer** with the same *VAT*. Finally, I invoke **getAllCustomers** again, and iterate through the list, ensuring that the customer is present in it.



**Test Case 5 – After deleting a certain customer, its sales should be removed from the database**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***salesRemovedOnCustomerRemoveTest ()*** *function)*

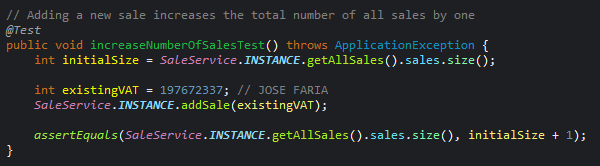
For this test, I start by choosing an existing *VAT* that was already present in the database, then use an instance of **CustomerService** to invoke **removeCustomer** with the chosen *VAT*. After that, I use an instance of **SaleService** to invoke **getSaleByCustomerVat** with the chosen *VAT* and ensure that the retrieved customer’s sale list’s size is 0.



**Test Case 6 – Adding a new sale increases the total number of all sales by one**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***increaseNumberOfSales ()*** *function)*

For this test, I start by using an instance of **SaleService** to invoke **getAllSales** and save the size of the returned list. Then, I choose an existing *VAT* that was already present in the database, then use an instance of **SaleService** to invoke **addSale** with the chosen *VAT*. After that, I use an instance of **SaleService** to invoke **getAllSales** again and ensure that the returned list’s size is equal to the initial saved size plus one.

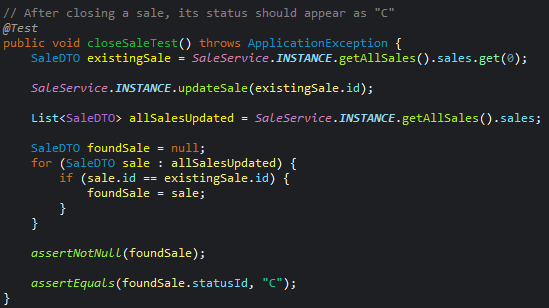


**Test Case 7 (Sale EXTRA Test 1) – After closing a sale, its status should appear as "C"**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***closeSaleTest ()*** *function)*

For this test, I start by using an instance of **SaleService** to invoke **getAllSales** and save the first sale. Then, I use an instance of **SaleService** to invoke **updateSale** with the chosen sale’s *ID*. After that, I use an instance of **SaleService** to invoke **getAllSales** again and iterate through the returned list, ensuring that:

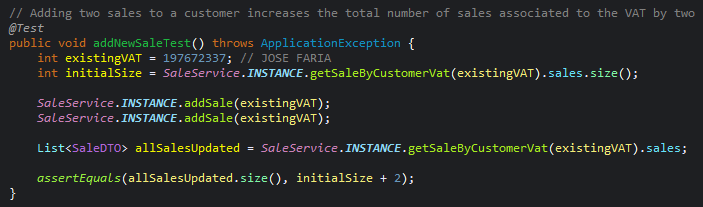
1. The chosen sale is on the list;
2. Its *status* *ID* is “*C*”, for closed.



**Test Case 8 (Sale EXTRA Test 2) – Adding two sales to a customer increases the total number of sales associated to the VAT by two**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***addNewSaleTest ()*** *function)*

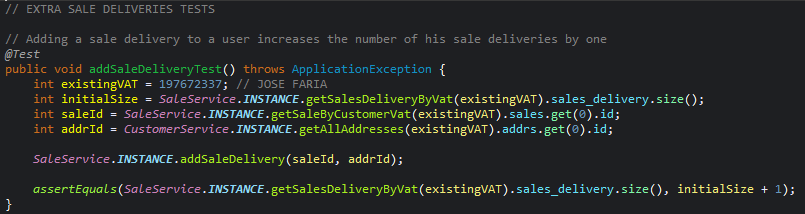
For this test, I start by choosing a *VAT* that exists in the database, then I use an instance of **SaleService** to invoke **getSaleByCustomerVat** with the chosen *VAT*, saving the size of the returned list. After this, I use an instance of **SaleService** to invoke **addSale** with the chosen *VAT* two times. Finally, I use an instance of **SaleService** to invoke **getSaleByCustomerVat** with the chosen *VAT* again, ensuring that the returned list’s size is equal to the original list’s size plus two.



**Test Case 9 (Sale Delivery EXTRA Test 1) – Adding a sale delivery to a user increases the number of his sale deliveries by one**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***addSaleDeliveryTest ()*** *function)*

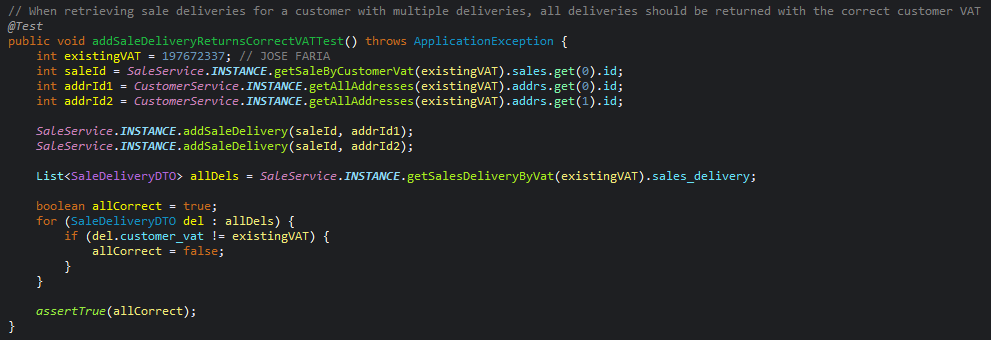
For this test, I start by choosing a *VAT* that exists in the database, then I use an instance of **SaleService** to invoke **getSalesDeliveryByVat** with the chosen *VAT*, saving the size of the returned list. After this, I use an instance of **SaleService** to invoke **getSaleByCustomerVat** with the chosen *VAT*, saving the first entry’s *ID*. Then, I use an instance of **SaleService** to invoke **getAllAdresses** with the chosen *VAT*, saving the first entry’s *ID*. Finally, I use an instance of **SaleService** to invoke **addSaleDelivery** with the saved sale *ID* and address *ID*, ensuring that, when **getSalesDeliveryByVat** is executed again, the returned list’s size is equal to the original list’s size plus one.



**Test Case 10 (Sale Delivery EXTRA Test 2) – When retrieving sale deliveries for a customer with multiple deliveries, all deliveries should be returned with the correct customer VAT**

*(The code for this test case is present in the* ***src/test/java/vvs\_DBSetup/TestsDBSetup.java*** *file, in the* ***addSaleDeliveryReturnsCorrectVATTest ()*** *function)*

For this test, I start by choosing a *VAT* that exists in the database, then I use an instance of **SaleService** to invoke **getSaleByCustomerVat** with the chosen *VAT*, saving the first entry’s *ID*. Then, I use an instance of **SaleService** to invoke **getAllAdresses** with the chosen *VAT*, saving the first and second entries’ *ID*. After that, I use an instance of **SaleService** to invoke **addSaleDelivery** with the saved sale *ID* and both addresses’ *ID*. Finally, I use an instance of **SaleService** to invoke **getSalesDeliveryByVat** with the chosen *VAT*, iterating through the list and ensuring that all deliveries are associated to the chosen *VAT*.



**Mockito Tests**

**Analysis of SUT Mockability:**

After analyzing the System Under Test (SUT), I've identified several architectural constraints that prevent direct use of Mockito for testing the business layer modules:

1. **Singleton Pattern Implementation**: The service classes (**CustomerService** and **SaleService**) are implemented as *Java* *Enums* with *Singleton* instances (e.g., **CustomerService.INSTANCE**), which cannot be directly instantiated or extended for testing.
2. **Direct Instantiation of Dependencies**: Dependencies are directly instantiated inside methods rather than being injected. For example:

A black screen with colorful text

AI-generated content may be incorrect.

This tight coupling makes it impossible to substitute these dependencies with mocks.

1. **No Dependency Injection:** There's no mechanism to inject mock objects for testing, as dependencies are created on-demand within methods.
2. **No Interface Abstractions:** The codebase lacks interfaces for key components, making it difficult to create and inject test doubles.

**Required Refactoring for Mockito Testing:**

To enable proper testing with **Mockito**, I've implemented the following refactoring approach:

1. **Created interfaces for dependencies:** Define interfaces for components that need to be mocked in tests.
2. **Added dependency injection mechanisms:** Modified service classes to accept dependencies through setters or constructors.
3. **Maintained backward compatibility:** Ensured existing code continued to work by providing default implementations.
4. **Extracted direct instantiations:** Moved direct instantiations to fields or constructors to enable substitution.

**Example Refactoring and Testing with Mockito:**

I've refactored part of the project to demonstrate this approach, focusing on the **CustomerService** and its dependency on **CustomerFinder**.

1. A screenshot of a computer program

   AI-generated content may be incorrect.**Created an interface for CustomerFinder:**
2. **Implemented the interface in CustomerFinder:**





1. A computer screen with text on it

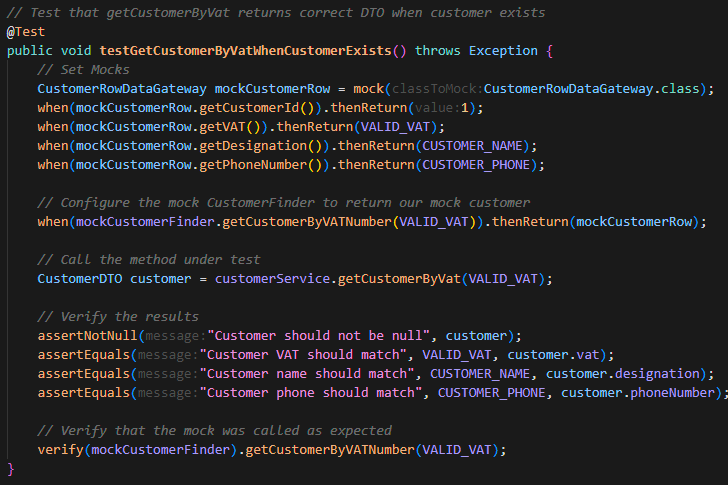
   AI-generated content may be incorrect.**Refactored CustomerService to support Dependency Injection:**

**(The methods now use the injected dependency instead of direct instantiation)**

1. **Created a Mockito Test for CustomerService:**

A computer screen shot of a program

AI-generated content may be incorrect.



**Benefits of Refactoring:**

This refactoring approach offers several benefits:

1. **Testability:** Services can now be tested in isolation using Mockito to mock their dependencies;
2. **Minimal Changes:** The existing API remains unchanged, ensuring backward compatibility;
3. **Singleton Preservation:** The singleton pattern is maintained while enabling testability;
4. **Reduced Coupling:** The code is now less tightly coupled and more modular.

**Conclusions:**

The original architecture of the SUT's business layer made it impossible to use **Mockito** effectively for unit testing. By introducing interfaces and dependency injection mechanisms, I've made the code testable while maintaining its original structure and behavior.

This approach allows for proper isolation of components during testing and improves the overall design of the system without requiring complete rewrite. The refactoring demonstrates how even legacy code with *Singletons* and *Tight Coupling* can be adapted for effective unit testing with minimal changes to the core architecture.

**Faults registered at Backlog**

Besides the two bugs listed in the **HtmlUnit** **Tests** section, when using the application normally I found three more bugs, that are also published as issues in the backlog project.

**Link to the project:** <https://vvs2425-01.backlog.com/projects/VVS_PROJ2_01>

**(Managed by Sara Canhoto – fc43177)**

**Bug 1:** **It is possible to add a sale to a non-existing customer**

**Bug Location:**

The bug is in ***src/main/java/webapp/services/SaleService.java*** in the **addSale** method. This method allows adding a sale to a customer by only validating if the VAT number is valid, without checking if the customer exists in the database.

A computer screen shot of a computer

AI-generated content may be incorrect.

**Rip Model:**

1. **Reach**: The fault can be reached when a user enters a valid *VAT* number (one that passes the **isValidVAT** check) but corresponds to a non-existent customer.
2. **Infection**: The infection occurs when a sale is created with a customer *VAT* that doesn't exist in the database. The database allows this because there's no foreign key constraint between the **SALE** table and the **CUSTOMER** table in the database schema.
3. **Propagation**: The error propagates when:

* A sale is successfully added to the database with a non-existent customer *VAT*;
* Later operations that assume a customer exists for every sale will fail or produce incorrect results;
* Data integrity is compromised as orphaned sales exist in the system.

Looking at the database schema in **createDDLHSQLDB.sql**, we can see that the **CUSTOMER\_VAT** column in the **SALE** table is not defined as a foreign key referencing the **VATNUMBER** in the **CUSTOMER** table.

**Correction Effort Estimate:**

The correction for this bug would require:

1. **Code Changes**:

* Modify the **addSale** method in **SaleService.java** to check if the customer exists before adding a sale;
* This would involve using the **CustomerFinder** class to verify the customer's existence.

1. **Database Changes**:

* Add a foreign key constraint to the database schema to enforce referential integrity.

**Estimated effort:**

* **Code Changes:** Low effort (1-2 hours)
* **Testing:** Medium effort (2-4 hours)
* **Database Schema Changes:** Medium effort (2-4 hours)

**Fixes:**

Modified the **addSale** method in **SaleService.java** to check if the customer exists before creating a sale:A computer code with text

AI-generated content may be incorrect.

(I also changed the database schema in **createDDLHSQLDB.sql** to the one explained in the **HtmlUnit** **Tests** section)

**Bug 2:** **It is possible to introduce scripts in the address field when inserting a new address to a customer**

**Bug Location:**

The vulnerability is in ***src/main/java/webapp/webpresentation/GetCustomerPageController.java***, where user input for addresses is directly concatenated without any sanitization:



Additionally, in ***src/main/webapp/CustomerInfo.jsp***, the address data is directly output without any escaping:

A screen shot of a computer code

AI-generated content may be incorrect.

**RIP Model:**

1. **Reach**: The fault can be reached when a user submits an address containing malicious script tags through the "Insert new Address to Customer" form.
2. **Infection**: The infection occurs when:

* The unsanitized input containing script tags is stored in the database;
* The application fails to escape HTML special characters when displaying the address.

1. **Propagation**: The error propagates when:

* The stored malicious script is retrieved from the database;
* The script is rendered in the browser when viewing the customer information;
* The browser executes the script, potentially stealing cookies, session tokens, or performing other malicious actions.

**Database Schema Impact:**

Looking at the database schema in **createDDLHSQLDB.sql**, the **ADDRESS** table has an **ADDRESS CHAR(100)** column that stores the concatenated address parts without any validation or sanitization.

**Correction Effort Estimate:**

The correction for this vulnerability would require:

1. **Input Sanitization**:

* Modify the GetCustomerPageController.java to sanitize user input before storing it;
* Implement HTML escaping for special characters like <, >, ", ', etc.

1. **Output Encoding**:

* Update **CustomerInfo.jsp** to properly encode output using ***JSTL's <c:out>*** tags instead of direct **EL** expressions.

**Estimated effort:**

* **Code changes:** Low to Medium effort (2-4 hours)
* **Testing:** Medium effort (2-4 hours)
* **Security Validation:** Medium effort (2-4 hours)

**Fixes:**

* In **GetCustomerPageController.java**, the controller needs to sanitize the input before storing it in the database:

A screenshot of a computer program

AI-generated content may be incorrect.

* In **CustomerInfo.jsp**, replaced direct **EL** expressions with ***JSTL's <c:out>*** tags to properly escape output:

A computer code with black text

AI-generated content may be incorrect.

**Bug 3:** **Cannot update the total of a sale**

**Bug Location:**

The bug is in multiple files that collectively prevent updating the total of a sale:

1. A white background with black text

   AI-generated content may be incorrect.In ***src/main/java/webapp/services/SaleDTO.java***, the constructor ignores the provided total parameter and always sets it to *0.0*:
2. A screenshot of a computer code

   AI-generated content may be incorrect.In ***src/main/java/webapp/services/SaleService.java***, there's no method to update the total of a sale, only to update its status:
3. In ***src/main/java/webapp/persistence/SaleRowDataGateway.java***, the SQL update statement only updates the status field, not the total:

A screen shot of a computer code

AI-generated content may be incorrect.

**Root Cause Analysis (RIP Model)**

1. **Reach**: The fault can be reached when a user attempts to update the total of a sale. However, there's no UI component or service method to perform this operation.
2. **Infection**: The infection occurs in multiple places:

* The **SaleDTO** constructor hardcodes the total to 0.0, ignoring any provided value;
* The **SaleService** lacks a method to update the total;
* The **SaleRowDataGateway**'s **updateSale** method only updates the status, not the total;
* The database schema supports storing a total value, but the application doesn't utilize it properly.

1. **Propagation**: The error propagates when:

* Sales are created with a total of *0.0*;
* The UI displays this value (*0.0*) for all sales;
* Users have no way to update the total through the application;
* The total field becomes effectively useless as it always shows 0.0.

**Database Schema Impact:**

The database schema in **createDDLHSQLDB.sql** correctly includes a TOTAL field in the **SALE** table, however, the application code doesn't properly utilize this field.

**Correction Effort Estimate:**

The correction for this issue would require:

1. **Code Changes**:

* Fix the **SaleDTO** constructor to use the provided total value;
* Add methods to **SaleService** and **SaleRowDataGateway** to update the total;
* Create a new controller and *JSP* page for updating sale totals;
* Update the *UI* to allow users to input and modify totals.

1. **Testing**:

* Unit tests for the new functionality;
* Integration testing to ensure proper database updates;
* UI testing for the new form.

**Estimated effort:**

* **Code changes:** Medium effort (4-6 hours)
* **Testing:** Medium effort (2-4 hours)
* **UI Changes:** Medium effort (2-4 hours)
* **Total**: Medium to Large (8-14 hours)

**Conclusion**

Throughout this report, several key testing activities were successfully carried out to validate and improve the functionality of the web application under test. **HtmlUnit** was used to simulate realistic user interactions, enabling verification of core use cases such as customer registration, address management, sales processing, and delivery tracking. **DbSetup** facilitated controlled database initialization, making it possible to test business rules and data integrity scenarios consistently.

Additionally, critical bugs were identified and resolved, including issues related to table formatting, data orphaning, and input sanitization. The report also demonstrated how architectural constraints initially hindered the use of **Mockito** and how targeted refactoring - introducing interfaces and dependency injection - allowed for effective unit testing of business logic components.

Ultimately, this report achieved a comprehensive **validation of the system’s behavior**, **improved test coverage**, **enhanced** **testability**, and contributed meaningful **fixes** that strengthened the overall robustness and maintainability of the application.